

Fig. 1a

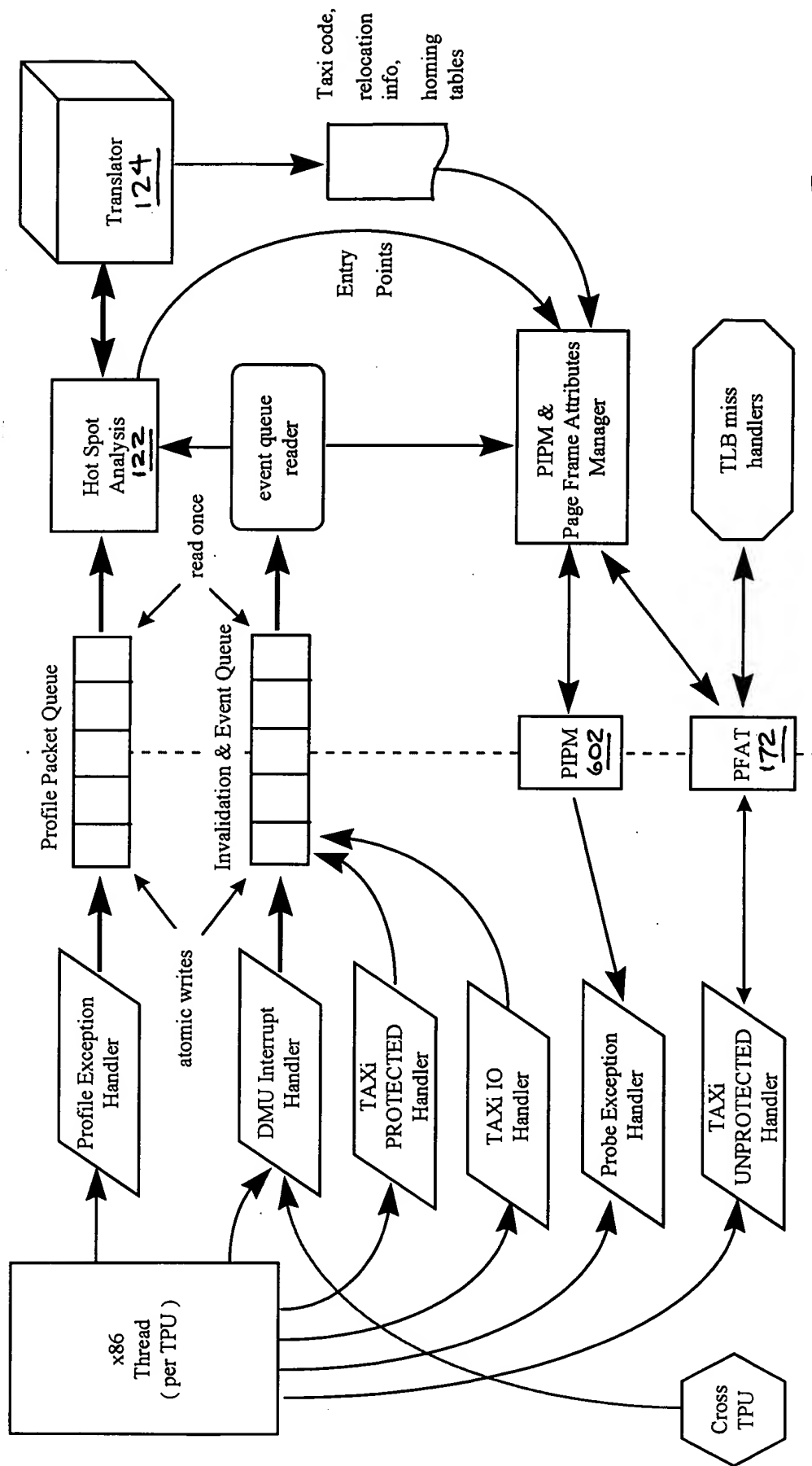
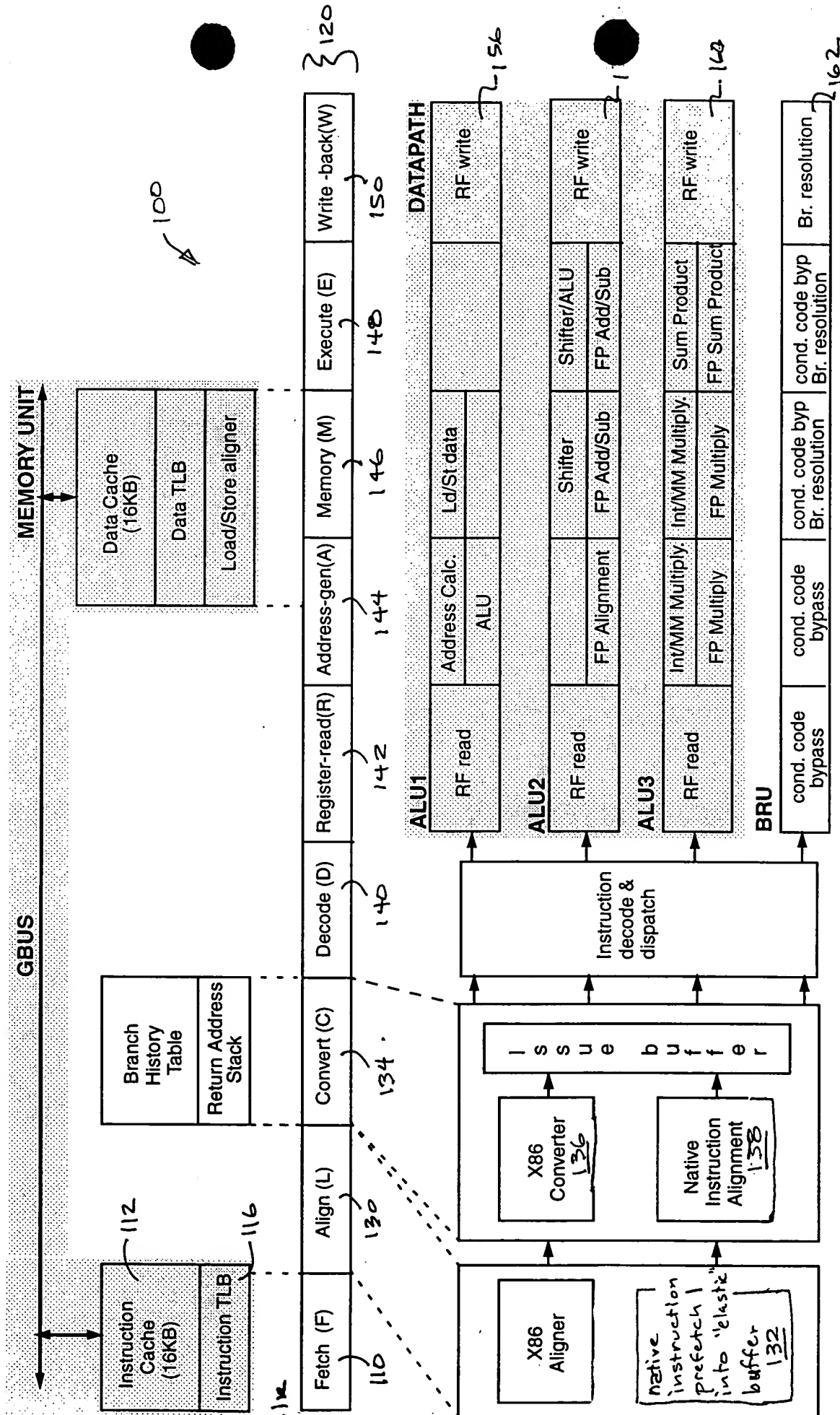


Fig. 1b

Fig. 1c



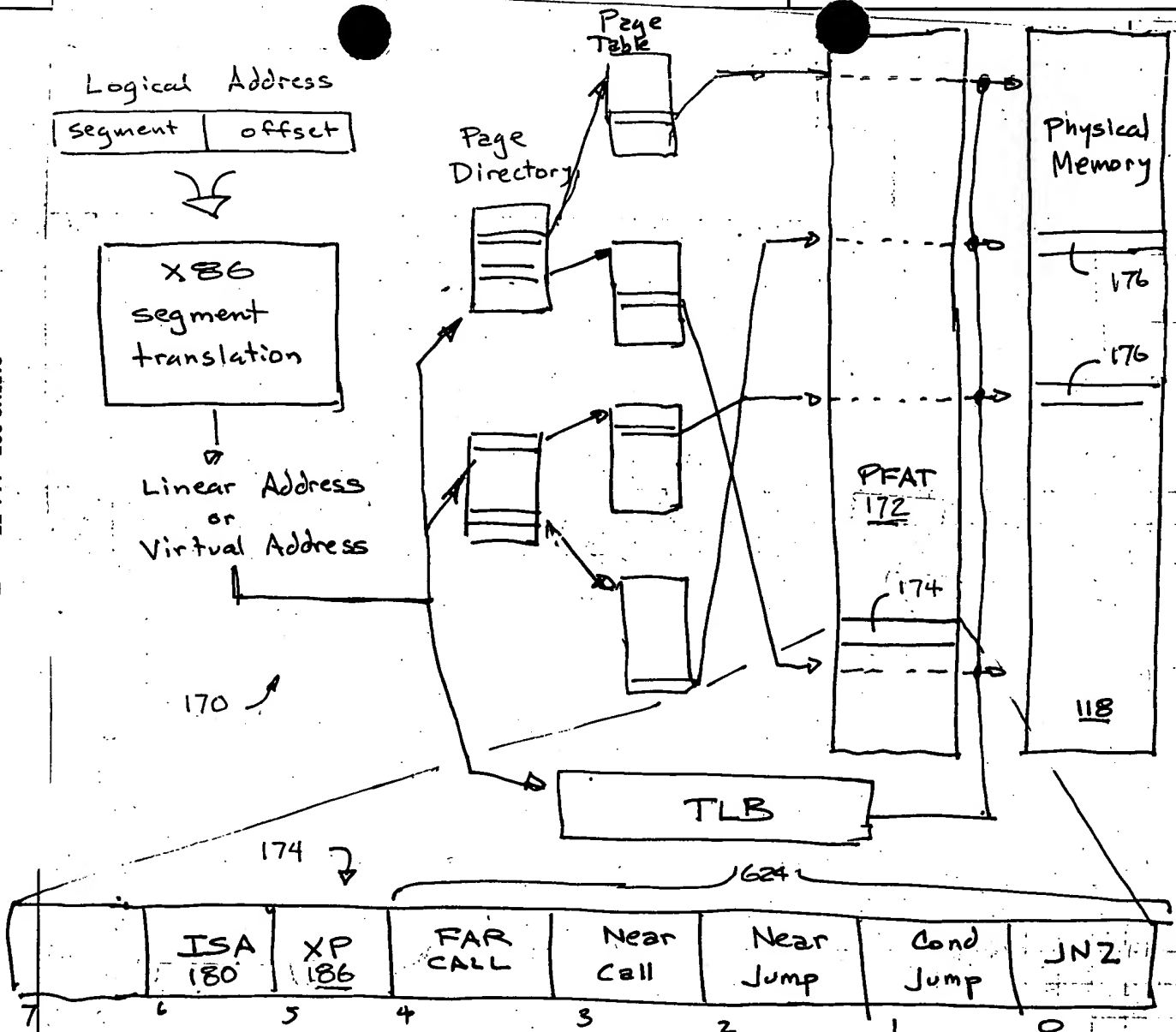


Fig. 1d - Memory Mapping

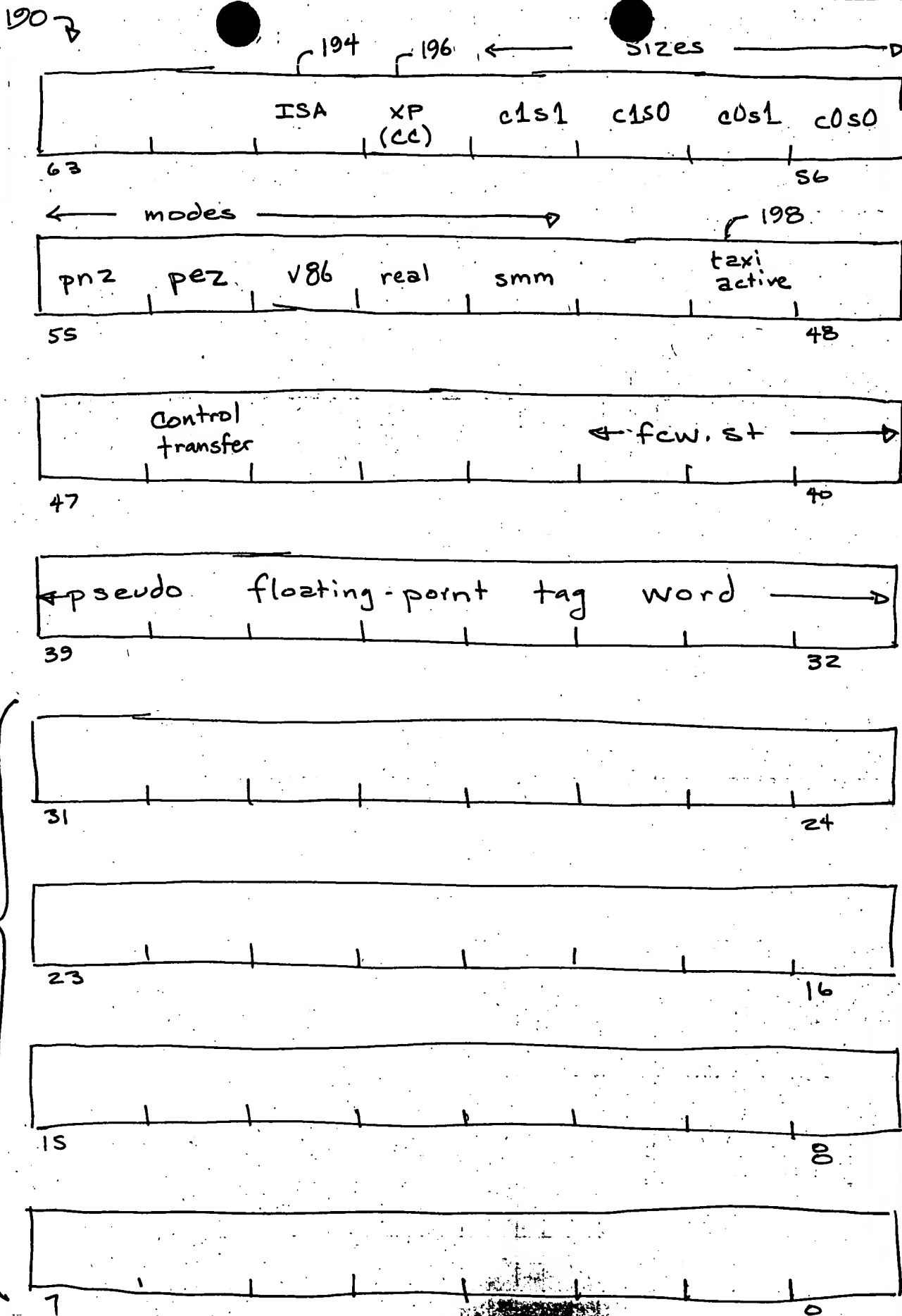


Fig. 1e

I-TLB property bits	Decoded property values			Interpretation	Instructions sent to:	Collect profile trace-packets	Probe for translated code	I/O memory reference exceptions
	ISA 194	CC 200	Protected					
00	Tap	Tap	no	Native code observing native RISCy calling conventions	Native decoder	No	No	Fault if SEG.tio
01	Tap	x86	no	Native code observing x86 calling conventions	Native decoder	No	No	Fault if SEG.tio
10	x86	x86	no	x86 code, unprotected - TAX! profile collection only	x86HW converter	If enabled	No	Trap if profiling
11	x86	x86	yes	x86 code, protected - TAX! code may be available	x86HW converter	If enabled	Based on I-TLB probe attributes	Trap if profiling

Fig. 2a Significance of the I-TLB property bits

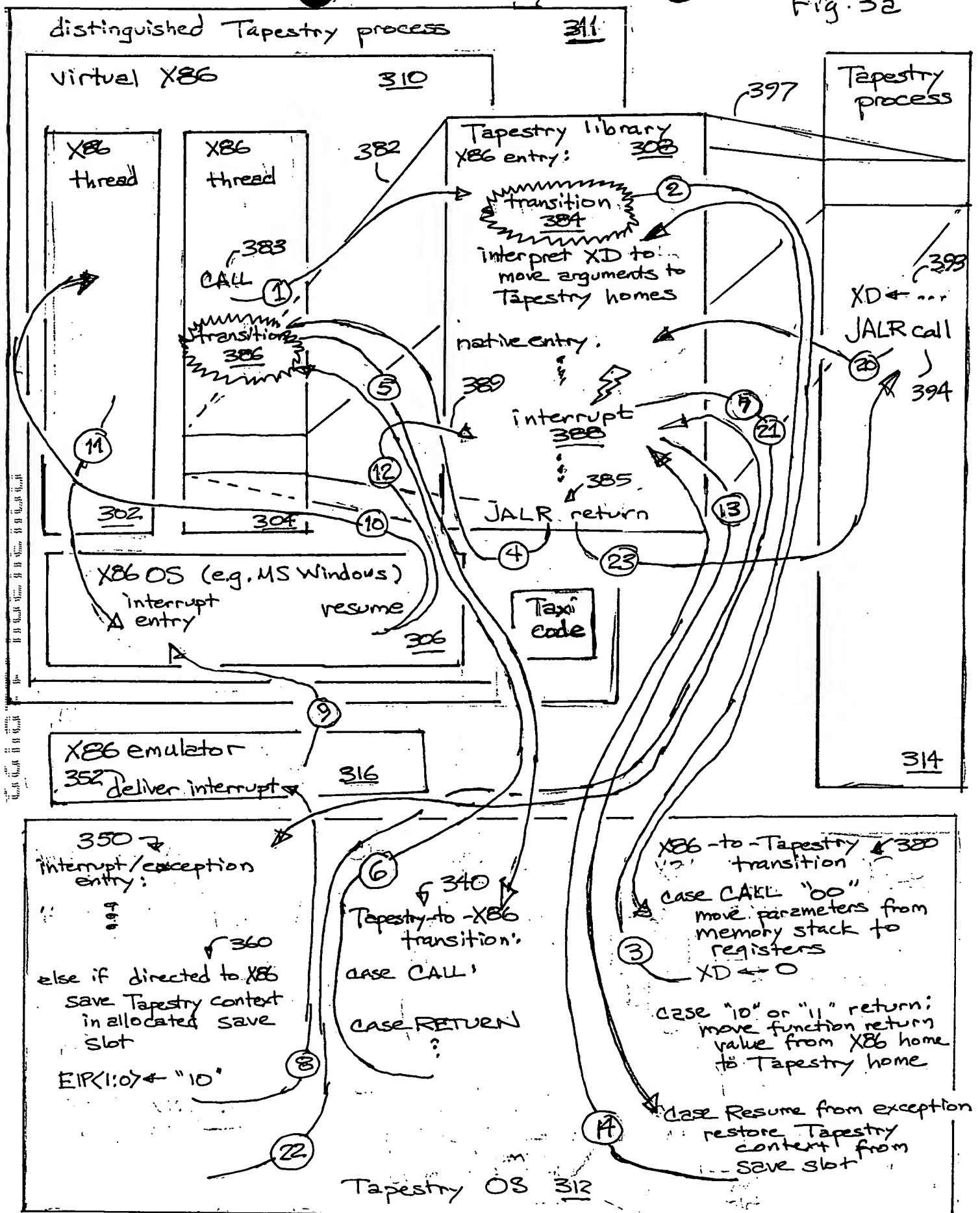
Transition (source => dest) ISA & CC property values	Handler Action
00 => 00	No transition exception
00 => 01	VECT_xxx_X86_CC exception - handler converts from native to x86 conventions
00 => 1x	VECT_xxx_X86_CC exception - handler converts from native to x86 conventions, sets up expected emulator and profiling state
01 => 00	VECT_xxx_TAP_CC exception - handler converts from x86 to native conventions
01 => 01	No transition exception
01 => 1x	VECT_X86_ISA exception [conditional based on PCW.X86_ISA_ENABLE flag] - sets up expected emulator and profiling state
1x => 00	VECT_xxx_TAP_CC exception - handler converts from x86 to native conventions
1x => 01	VECT_TAP_ISA exception [conditional based PCW.TAP_ISA_ENABLE flag] - no convention conversion necessary
1x => 10	No transition exception - [profile complete possible, probe possible]
1x => 11	No transition exception - [profile complete possible, probe NOT possible]

Fig. 2b ISA & CC transition exception flow

name	description	type
VECT_call_X86_CC	push args, return address, set up x86 state	fault on target instruction
VECT_jump_X86_CC	set up x86 state	fault on target instruction
VECT_ret_no_fp_X86_CC	return value to eax:edx, set up x86 state	fault on target instruction
VECT_ret_fp_X86_CC	return value to x86 fp stack, set up x86 state	fault on target instruction
VECT_call_TAP_CC	x86 stack args, return address to registers	fault on target instruction
VECT_jump_TAP_CC	x86 stack args to registers	fault on target instruction
VECT_ret_no_fp_TAP_CC	return value to RV0	fault on target instruction
VECT_ret_any_TAP_CC	return type unknown, setup RV0 and RVDV	fault on target instruction

Fig. 2c CC transition exceptions

Fig. 32



Flat 32-bit "Near" Address Space

Transparency:

- . x86 code adheres to traditional x86 stack-based conventions
- . RISC uses higher performance register-based conventions
- . Caller has no knowledge of callee's ISA
- . Callee has no knowledge of ISA to which it will return

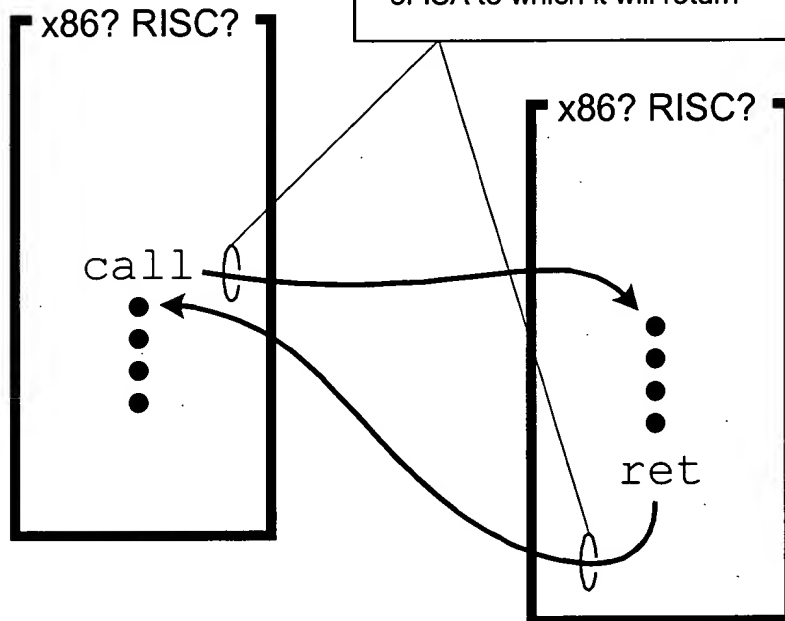


Fig. 3b

Flat 32-bit "Near" Address Space

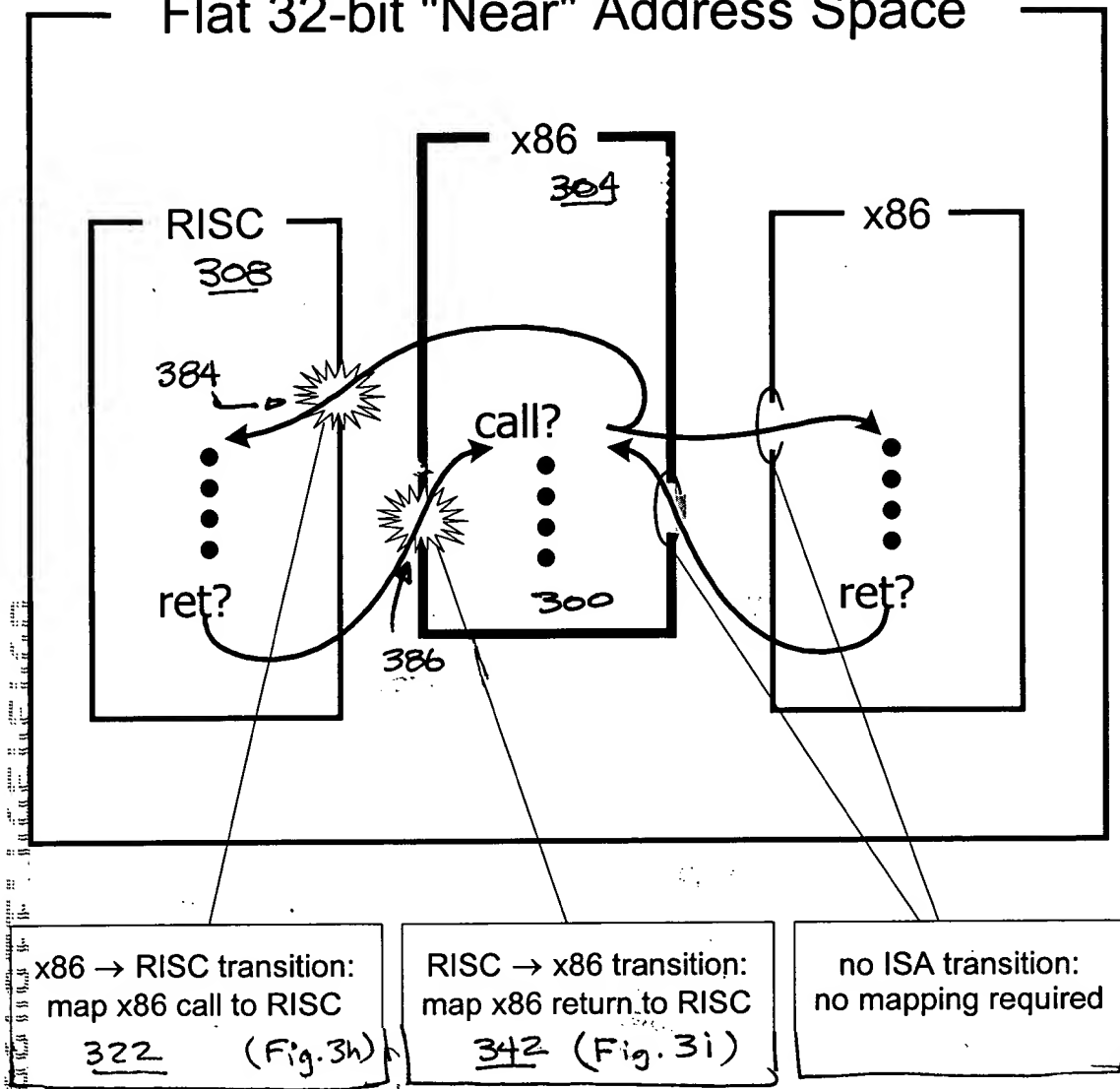


Fig. 3c

Flat 32-bit "Near" Address Space

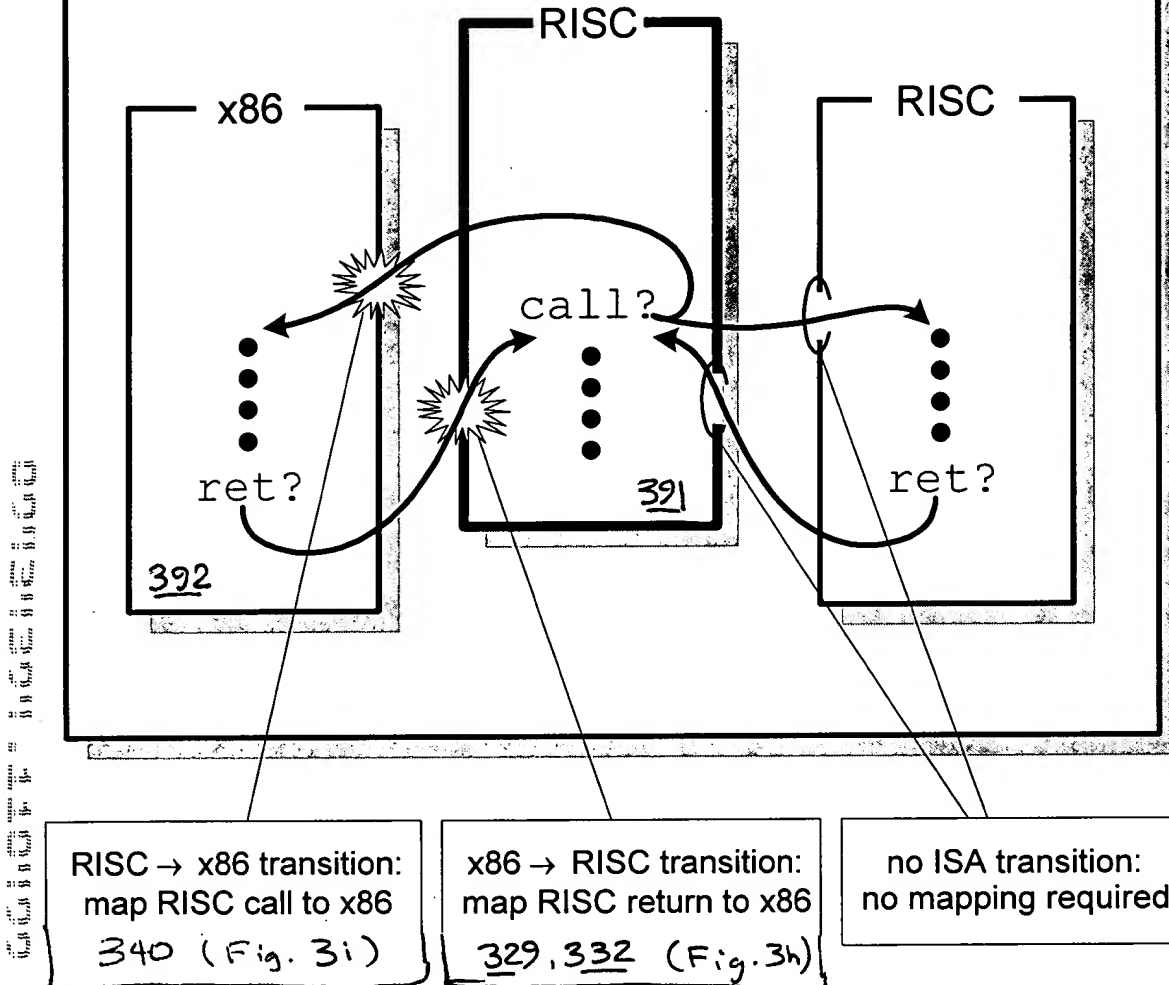


Fig. 3d

Flat 32-bit "Near" Address Space

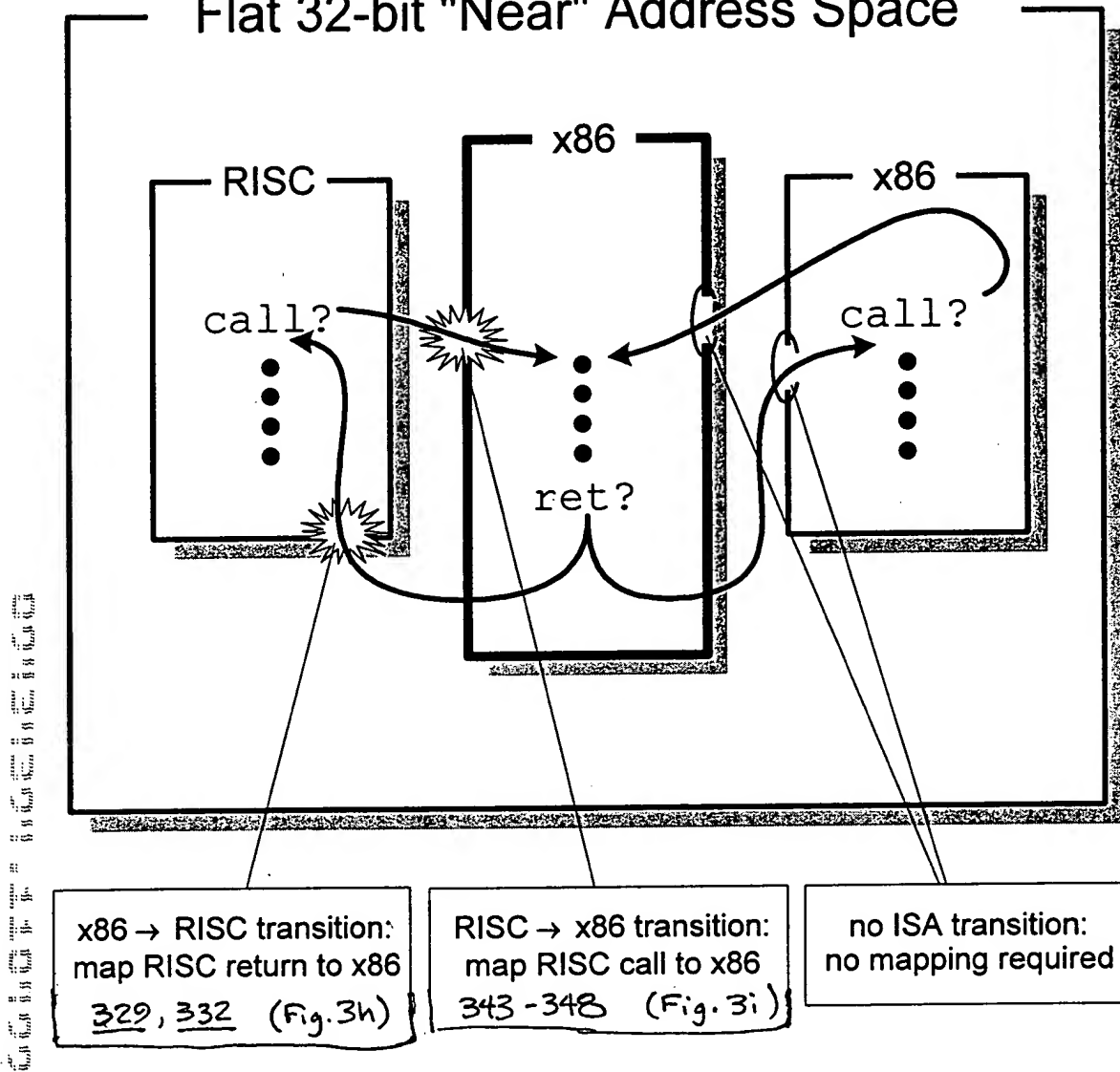


Fig. 3e

Flat 32-bit "Near" Address Space

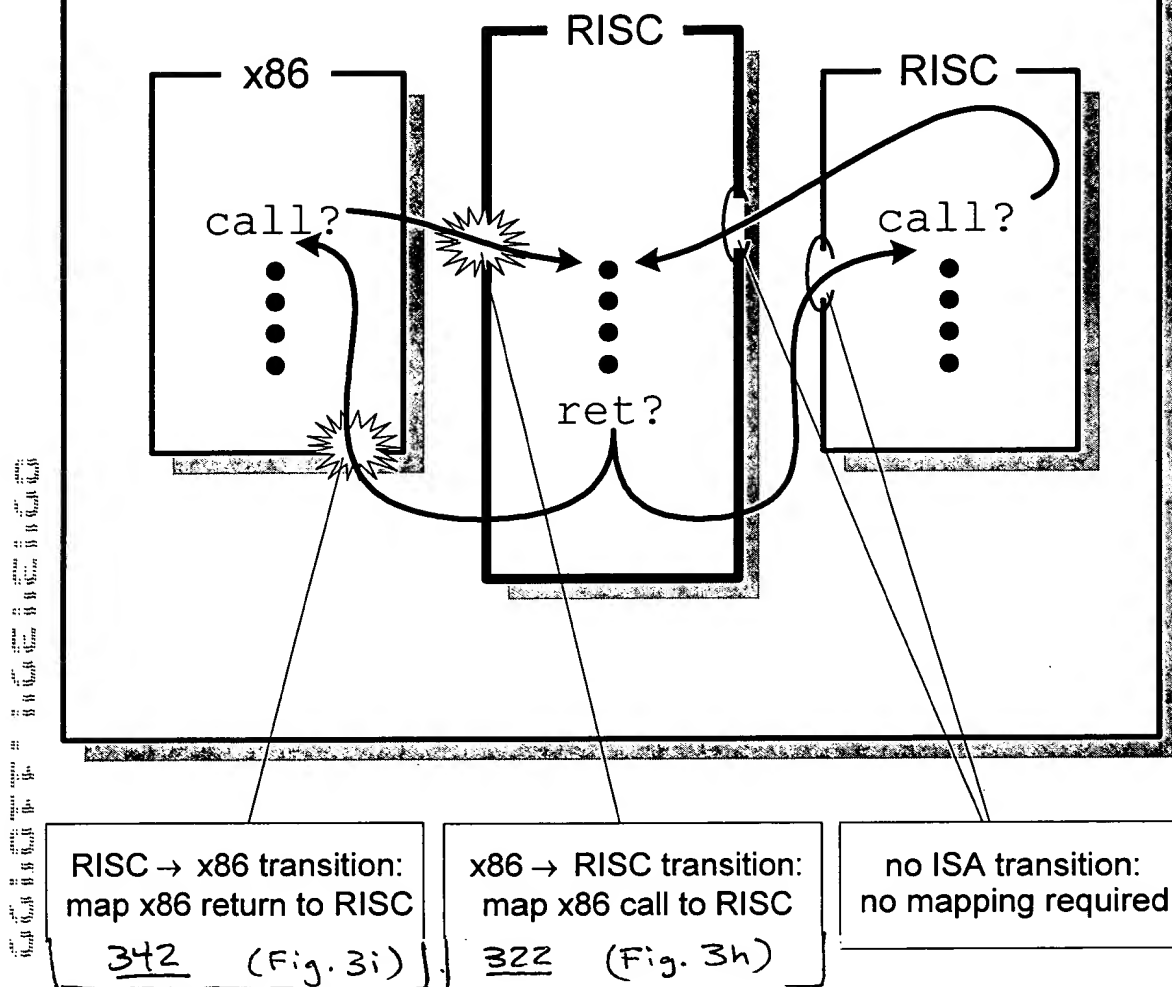


Fig. 3f

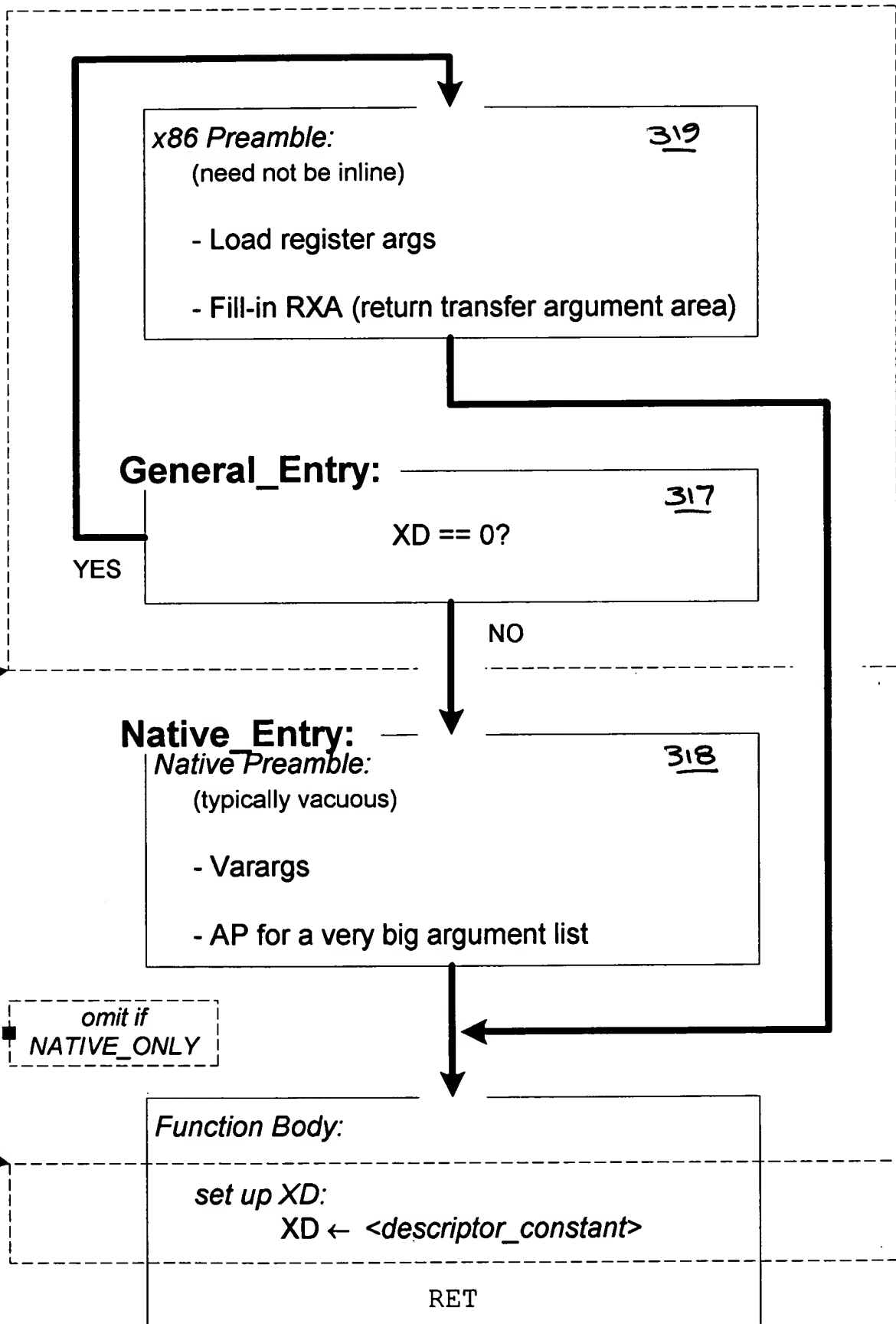


Fig. 3g

X86-to-Tapestry transition exception handler

// This handler is entered under the following conditions:

- // 1. An x86 caller invokes a native function
- // 2. An x86 function returns to a native caller
- // 3. x86 software returns to or resumes an interrupted native function following
- // an external asynchronous interrupt, a processor exception, or a context switch

```

dispatch on the two least-significant bits of the destination address {
case "00" // calling a native subprogram
    // copy linkage and stack frame information and call parameters from the memory
    // stack to the analogous Tapestry registers
    LR ← [SP++] // set up linkage register ~ 323
    AP ← SP // address of first argument ~ 324
    SP ← SP - 8 // allocate return transfer argument area ~ 326
    SP ← SP & (-32) // round the stack pointer down to a 0 mode 32 boundary ~ 327
    XD ← 0 // inform callee that caller uses X86 calling conventions ~ 328
case "01" // resuming an X86 thread suspended during execution of a native routine
    if the redundant copies of the save slot number in EAX and EDX do not match or if
        the redundant copies of the timestamp in EBX:ECX and ESI:EDI do not match {
        // some form of bug or thread corruption has been detected
        goto TAPESTRY_CRASH_SYSTEM( thread-corruption-error-code ) ~ 372
    }
    save the EBX:ECX timestamp in a 64-bit exception handler temporary register
        (this will not be overwritten during restoration of the full native context) ~ 373
    use save slot number in EAX to locate actual save slot storage ~ 374
    restore full entire native context (includes new values for all x86 registers) ~ 375
    if save slot's timestamp does not match the saved timestamp {
        // save slot as been reallocated; save slot exhaustion has been detected
        goto TAPESTRY_CRASH_SYSTEM( save-slot-overwritten-error-code ) ~ 377
    }
    free the save slot ~ 378
case "10" // returning from X86 callee to native caller, result already in registers
    RV0<63:32> ← edx<31:00> // in case result is 64 bits ~ 333
    convert the FP top-of-stack value from 80 bit X86 form to 64-bit form in RVDP ~ 334
    SP ← ESI // restore SP from time of call ~ 337
case "11" // returning from X86 callee to native caller, load large result from memory
    RV0..RV3 ← load 32 bytes from [ESI-32] // (guaranteed naturally aligned) ~ 330
    SP ← ESI // restore SP from time of call ~ 337
}
EPC ← EPC & -4 // reset the two low-order bits to zero ~ 336
RFE ~ 338

```

Fig. 3h

Tapestry-to-X86 transition exception handler

// This handler is entered under the following conditions:

// 1. a native caller invokes an x86 function

// 2. a native function returns to an x86 caller

switch on XD<3:0> { ~ 341

XD_RET_FP: // result type is floating point
F0/F1 ← FINFLATE.de(RVDP) // X86 FP results are 80 bits
SP ← from RXA save // discard RXA, pad, args
FPCW ← image after FINIT & push // FP stack has 1 entry
goto EXIT

XD_RET_WRITEBACK: // store result to @RVA, leave RVA in eax
RVA ← from RXA save // address of result area
copy decode(XD<8:4>) bytes from RV0..RV3 to [RVA]
eax ← RVA // X86 expects RVA in eax
SP ← from RXA save // discard RXA, pad, args
FPCW ← image after FINIT // FP stack is empty
goto EXIT

XD_RET_SCALAR: // result in eax:edx
edx<31:00> ← eax<63:32> // in case result is 64 bits
SP ← from RXA save // discard RXA, pad, args
FPCW ← image after FINIT // FP stack is empty
goto EXIT

XD_CALL_HIDDEN_TEMP: // allocate 32 byte aligned hidden temp
esi ← SP // stack cut back on return ~ 343
SP ← SP - 32 // allocate max size temp } 344
RVA ← SP // RVA consumed later by RR
LR<1:0> ← "11" // flag address for return & reload ~ 345
goto CALL_COMMON

default: // remaining XD_CALL_xxx encodings
esi ← SP // stack cut back on return ~ 343
LR<1:0> ← "10" // flag address for return ~ 346

CALL_COMMON:
interpret XD to push and/or reposition args ~ 347
[--SP] ← LR // push LR as return address

EXIT:
setup emulator context and profiling ring buffer pointer

}
RFE ~ 349 // to original target

Fig. 3i

interrupt/exception handler of Tapestry operating system:

// Control vectors here when a synchronous exception or asynchronous interrupt is to be
// exported to / manifested in an x86 machine.

// The interrupt is directed to something within the virtual X86, and thus there is a possibility
// that the X86 operating system will context switch. So we need to distinguish two cases:
// either the running process has only X86 state that is relevant to save, or
// there is extended state that must be saved and associated with the current machine context
// (e.g., extended state in a Tapestry library call in behalf of a process managed by X86 OS)
if execution was interrupted in the converter – EPC.ISA == X86 {
 // no dependence on extended/native state possible hence no need to save any } 351
 goto EM86_Deliver_Interrupt(interrupt-byte)
}
else if EPC.Taxi_Active {
 // A Taxi translated version of some X86 code was running. Taxi will rollback to an
 // x86 instruction boundary. Then, if the rollback was induced by an asynchronous external } 353
 // interrupt Taxi will deliver the appropriate x86 interrupt. Else, the rollback was induced
 // by a synchronous event so Taxi will resume execution in the converter, retriggering the
 // exception but this time will EPC.ISA == X86
 goto TAXi_Rollback(asynchronous-flag, interrupt-byte)
}
else if EPC.EM86 {
 // The emulator has been interrupted. In theory the emulator is coded to allow for such } 354
 // conditions and permits re-entry during long running routines (e.g. far call through a gate)
 // to deliver external interrupts
 goto EM86_Deliver_Interrupt(interrupt-byte)
}
else {
 // This is the most difficult case – the machine was executing native Tapestry code on
 // behalf of an X86 thread. The X86 operating system may context switch. We must save
 // all native state and be able to locate it again when the x86 thread is resumed.
 361
 allocate a free save slot; if unavailable free the save slot with oldest timestamp and try again
 save the entire native state (both the X86 and the extended state) } 362
 save the X86 EIP in the save slot
 overwrite the two low-order bits of EPC with “01” (will become X86 interrupt EIP) ~ 363
 store the 64-bit timestamp in the save slot, in the X86 EBX:ECX register pair (and, } 364
 for further security, store a redundant copy in the X86 ESI:EDI register pair)
 store the a number of the allocated save slot in the X86 EAX register (and, again for } 365
 further security, store a redundant copy in the X86 EDX register)
 goto EM86_Deliver_Interrupt(interrupt-byte) ~ 369
}

350 A

Fig. 3j


```

typedef struct {
    save_slot_t * newer;           // pointer to next-most-recently-allocated save slot } 379c
    save_slot_t * older;          // pointer to next-older save slot
    unsigned int64 epc;           // saved exception PC/IP
    unsigned int64 pcw;           // saved exception PCW (program control word) } 356
    unsigned int64 registers[63]; // save the 63 writeable general registers
    ...                          // other words of Tapestry context
    timestamp_t timestamp;        // timestamp to detect buffer overrun ~ 358
    int save_slot_ID;             // ID number of the save slot ~ 357
    boolean save_slot_is_full;    // full / empty flag ~ 359
} save_slot_t;

```

```

save_slot_t * save_slot_head;    // pointer to the head of the queue ~ 379a
save_slot_t * save_slot_tail;    // pointer to the tail of the queue ~ 379b

```

system initialization

reserve several pages of unpaged memory for save slots

Fig. 3k

Fig. 32-bit "Near" Address Space

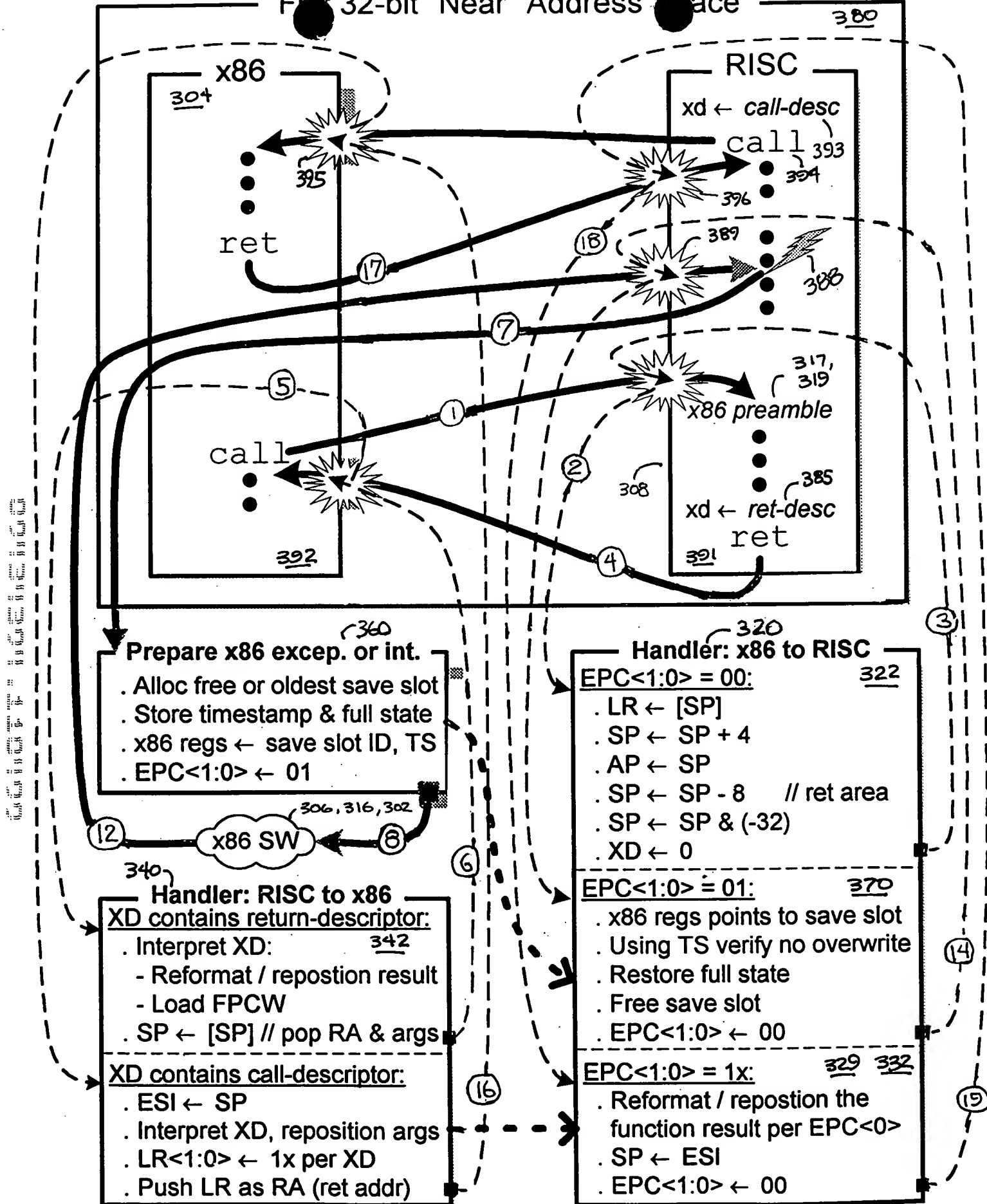


Fig. 31

Flat 32-bit "Near" Address Space

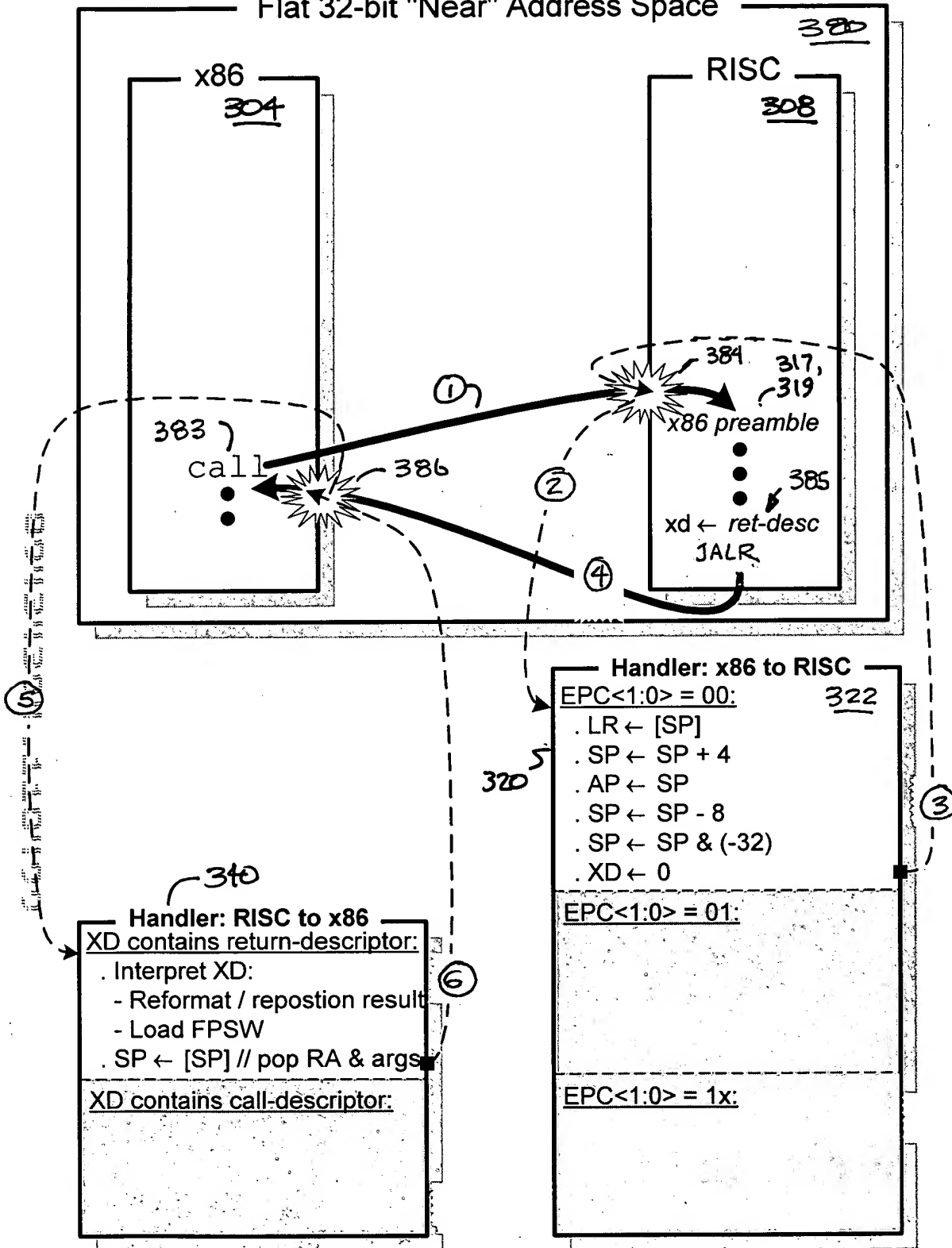


Fig. 3m

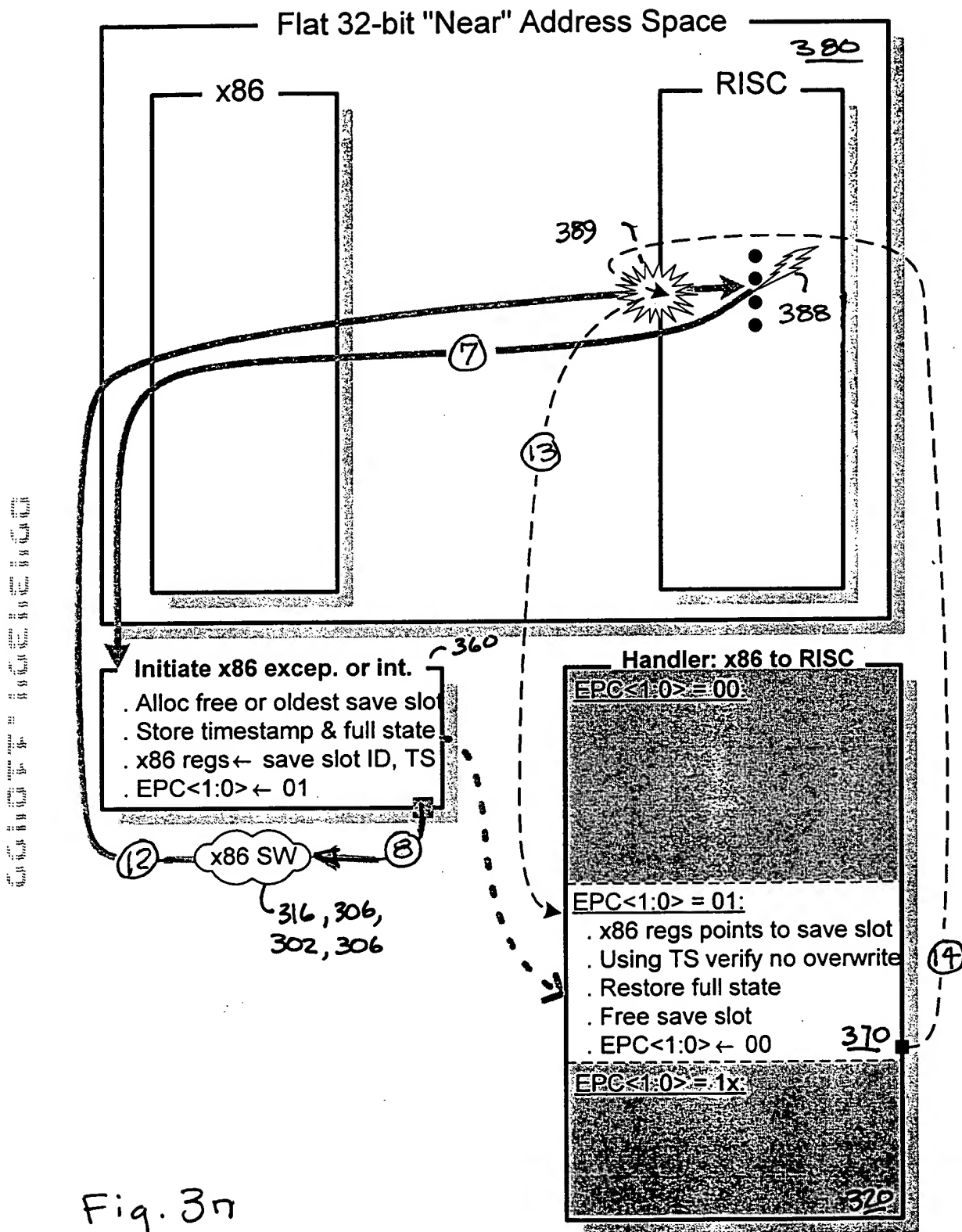


Fig. 3n

Flat 32-bit "Near" Address Space

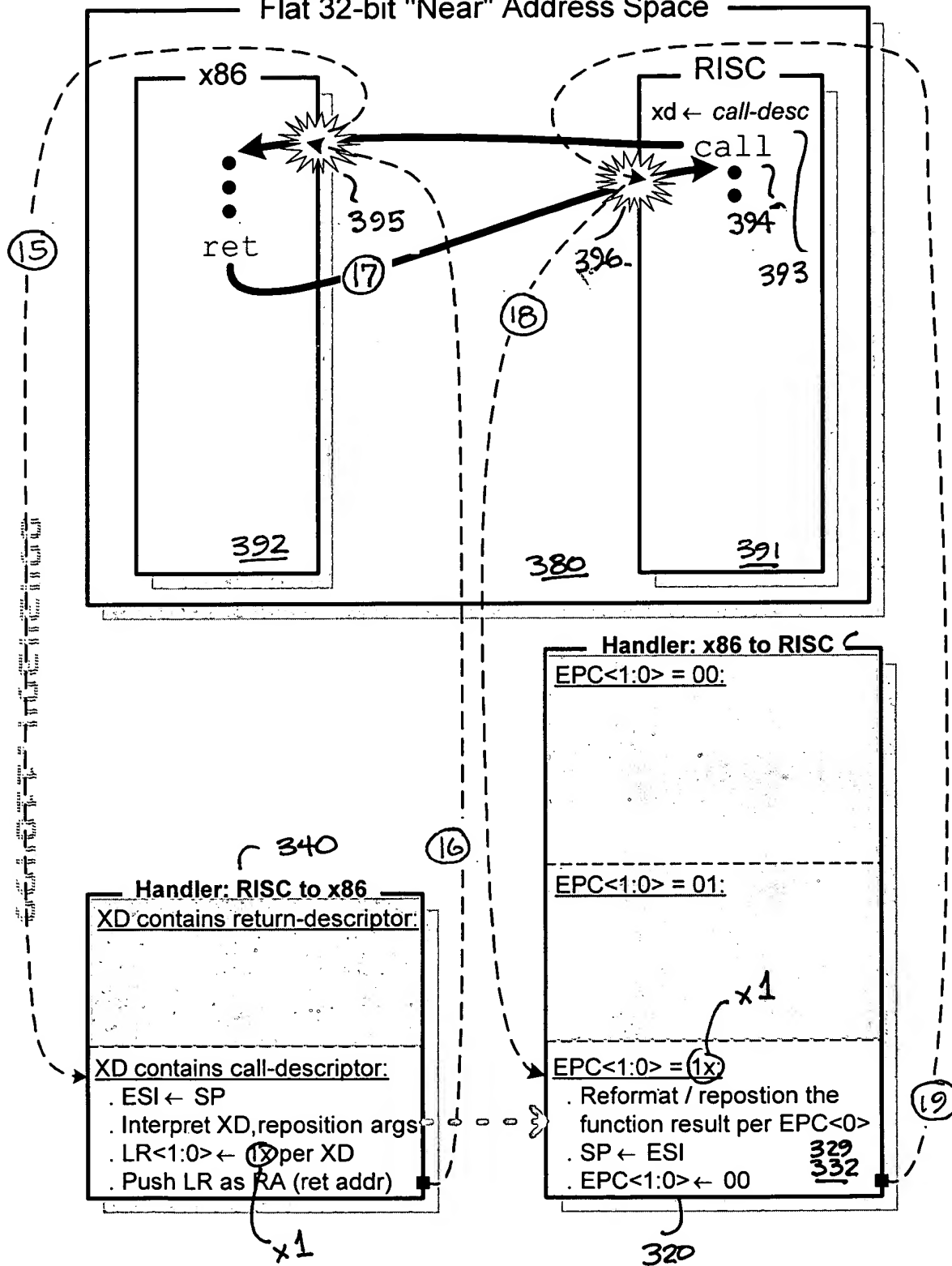
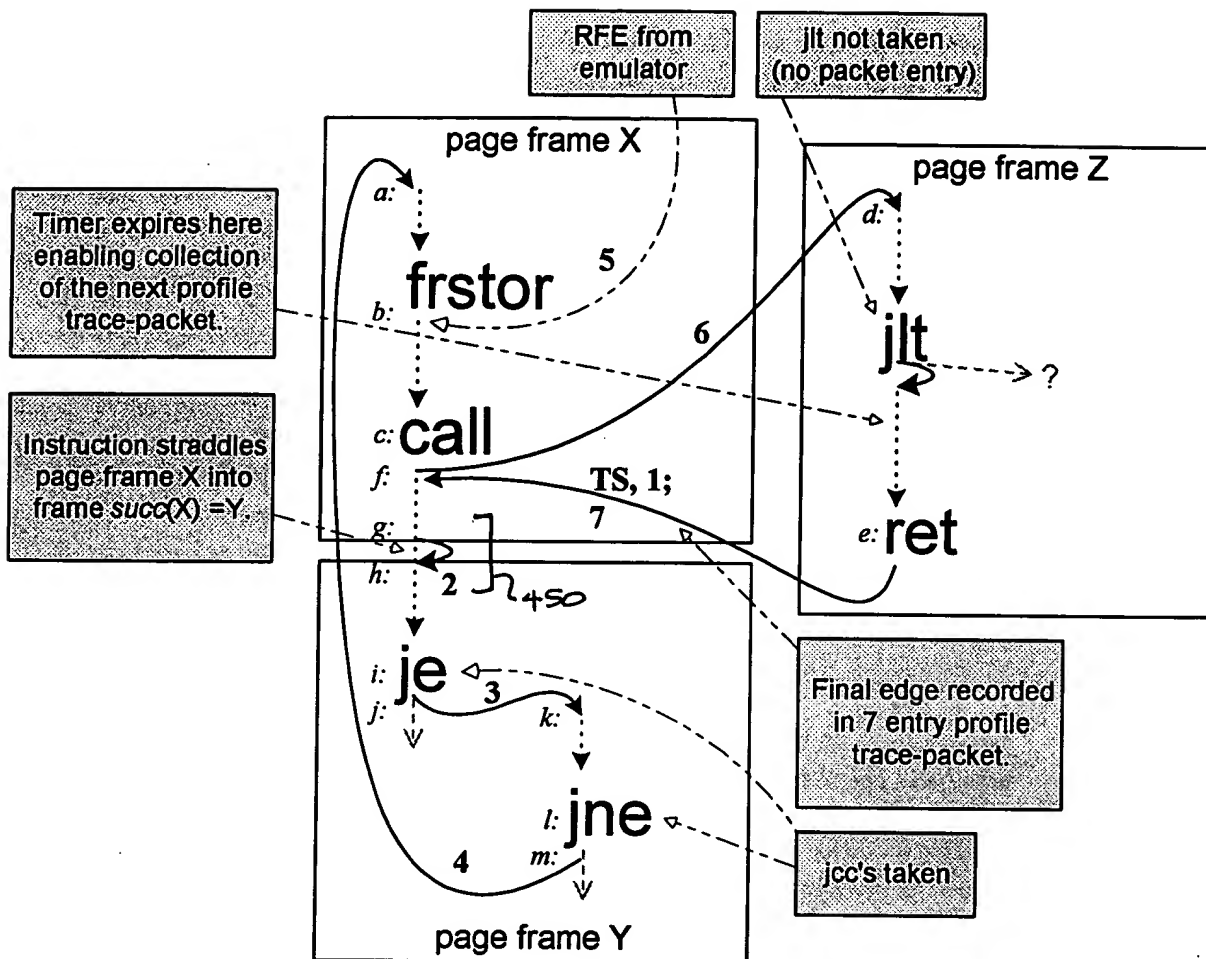


Fig. 30



7 entry trace packet

420 {

Entry	Event Code	Done Addr	Next Addr
64 bit time stamp			
1	ret	x86 context	phys X:f ~ 430
2	new page	phys Y:g	phys Y:h ~ 440, 454
3	jcc forward	phys Y:i	phys Y:k ~ 440
4	jnz backward	phys Y:l	phys X:a ~ 440
5	seq; env change	x86 context	phys X:b ~ 430
6	ip-rel near call	phys X:c	phys Z:d ~ 440
7	near ret	phys Z:e	phys X:f ~ 440

Fig. 4a

Source	Code	Event	Reuse event code	Profileable event	Initiate packet	Probeable event	Probe event bit - ITLB probe attribute or Emulator probe
RFE (Context at Point entry)	402 0.0000	Default (x86 transparent) event, reuse all converter values	yes	no no no no reuse event code	no no no no	no no no no	-
	0.0001	Simple x86 instruction completion (reuse event code)	yes				
	0.0010	Probe exception failed	yes				
	0.0011	Probe exception failed, reload probe timer	yes				
	0.0100	flush event	no	no	no	no	-
	0.0101	Sequential; execution environment changed - force event	no	yes	no	no	-
	0.0110	Far RET	no	yes	yes	no	-
	0.0111	IRET	no	yes	no	no	-
	0.1000	Far CALL	no	yes	yes	yes	Far call
	0.1001	Far JMP	no	yes	yes	no	-
	0.1010	Special; emulator execution, supply extra instruction data ^a	no	yes	no	no	-
	0.1011	Abort profile collection	no	no	no	no	-
	0.1100	x86 synchronous/asynchronous interrupt w/probe (GRP 0)	no	yes	yes	yes	Emulator probe
	0.1101	x86 synchronous/asynchronous interrupt (GRP 0)	no	yes	yes	no	-
	0.1110	x86 synchronous/asynchronous interrupt w/probe (GRP 1)	no	yes	yes	yes	Emulator probe
	0.1111	x86 synchronous/asynchronous interrupt (GRP 1)	no	yes	yes	no	-
Converter (Near_Edge entry)	1.0000	IP-relative JNZ forward (opcode: 75, 0F 85)	no	yes	yes	no	-
	1.0001	IP-relative JNZ backward (opcode: 75, 0F 85)	no	yes	yes	yes	Jnz
	1.0010	IP-relative conditional jump forward - (Jcc, Jcxz, loop)	no	yes	yes	no	-
	1.0011	IP-relative conditional jump backward - (Jcc, Jcxz, loop)	no	yes	yes	yes	Cond jump
	1.0100	IP-relative, near JMP forward (opcode: E9, EB)	no	yes	yes	no	-
	1.0101	IP-relative, near JMP backward (opcode: E9, EB)	no	yes	yes	yes	Near jump
	1.0110	RET/ RET imm16 (opcode C3, C2 /w)	no	yes	yes	no	-
	1.0111	IP-relative, near CALL (opcode: E8)	no	yes	yes	yes	Near call
	1.1000	REPE/REPNE CMPS/SCAS (opcode: A6, A7, AE, AF)	no	yes	no	no	-
	1.1001	REP MOVSB/STOS/LDOS (opcode: A4, A5, AA, AB, AC, AD)	no	yes	no	no	-
	1.1010	Indirect near JMP (opcode: FF /4)	no	yes	yes	no	-
	1.1011	Indirect near CALL (opcode: FF /2)	no	yes	yes	yes	Near call
	1.1100	load from I/O memory (TLB.asi != 0) { not used in T1 }	no	yes	no	no	-
	1.1101	available for expansion	no	no	no	no	-
	1.1110	Default converter event; sequential 406	no	no	no	no	-
	1.1111	New page (instruction ends on last byte of a page frame or straddles across a page frame boundary) 408	no	yes	no	no	-

a. Used by emulator for new x86 opcodes. Extra information supplied in Taxi_Control.special_opcode bits.

Fig. 4b

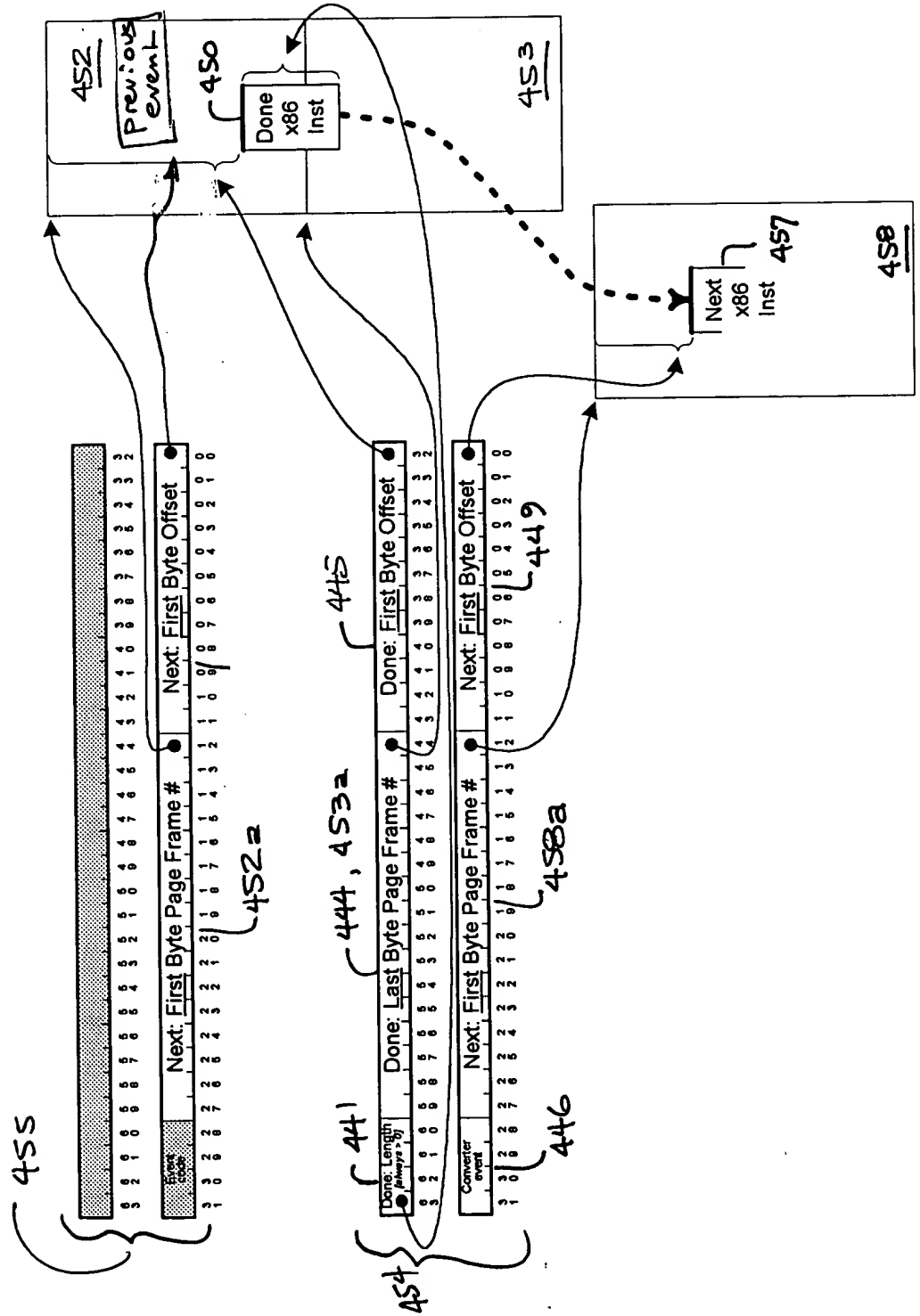


Fig. 4f

Events

pe_{init} = "initiate packet" profile event 418
 $pe_{\overline{init}}$ = non-"initiate packet" profile event 416
 pe = any profile event 416
 te = timer expiry
 ap = abort packet

State Variables

PR = Profile_Request flag 484
 PA = Profile_Active flag 482

Rules

$te\ event \Rightarrow$
 $PR \leftarrow 1$
 $PR \ \& \ \overline{PA} \ \& \ pe_{init} \Rightarrow$
 $PR \leftarrow 0$
 $PA \leftarrow 1$
Init Packet_Reg
Save timestamp
Log event (CAP)
Full packet? / Packet_Reg++
 $PA \ \& \ pe \Rightarrow$
Log event (CAP or NE)
Full packet? / Packet_Reg++
full packet \Rightarrow
 $PA \leftarrow 0$
Profile exception
 $ap\ event \Rightarrow$
 $PA \leftarrow 0$

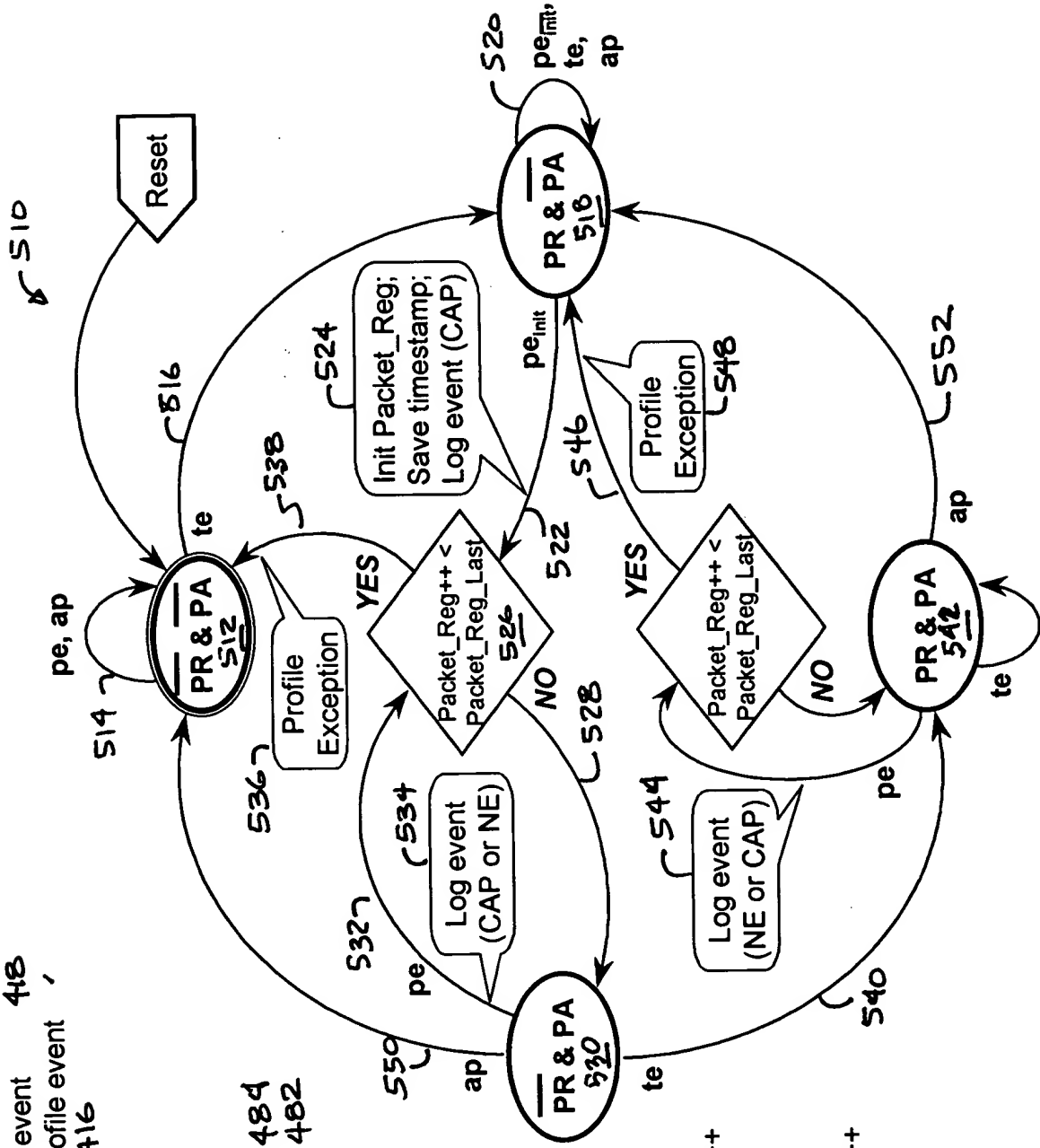


Fig. 5a

taxi profile entry generation

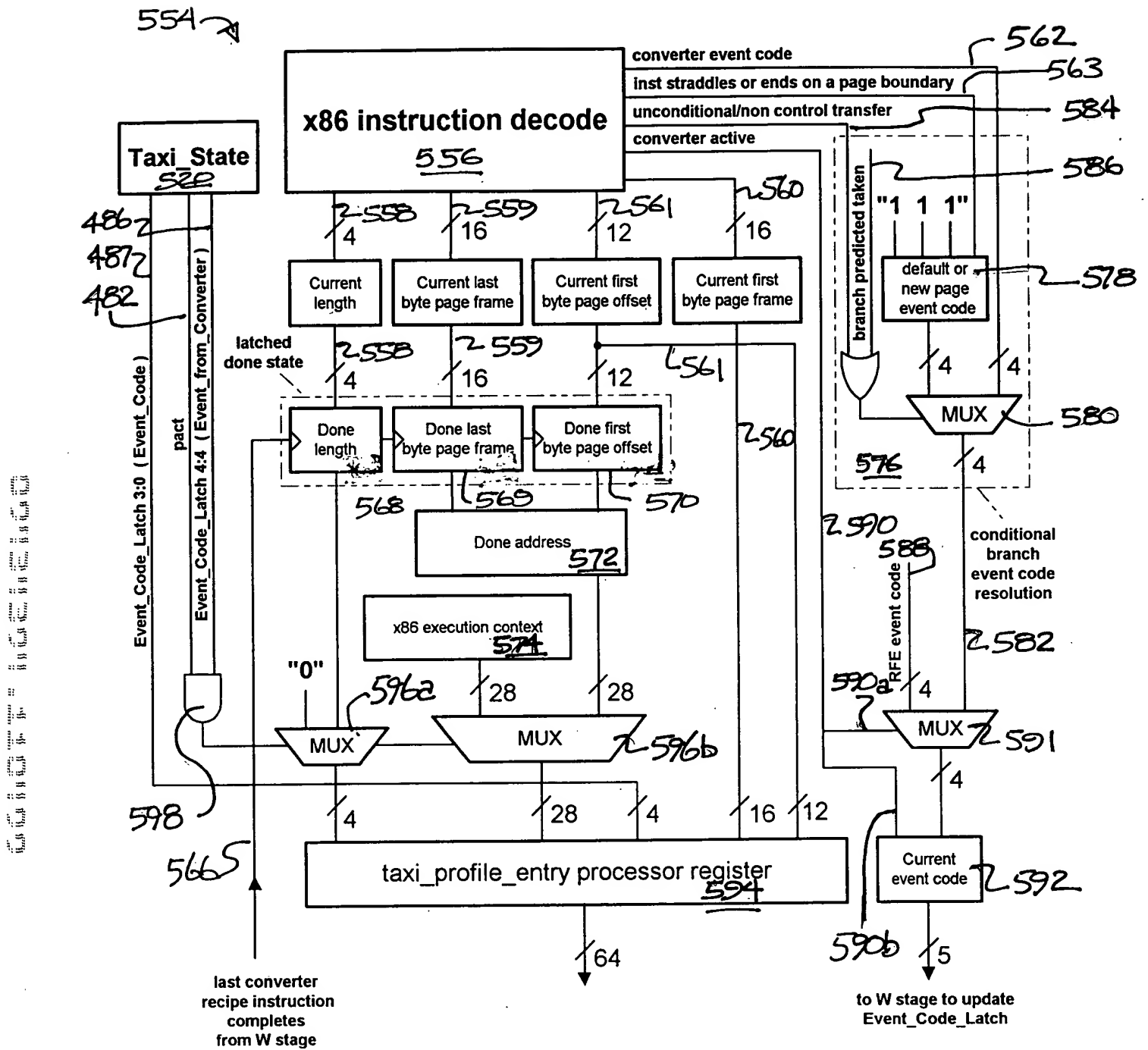


Fig. 5b

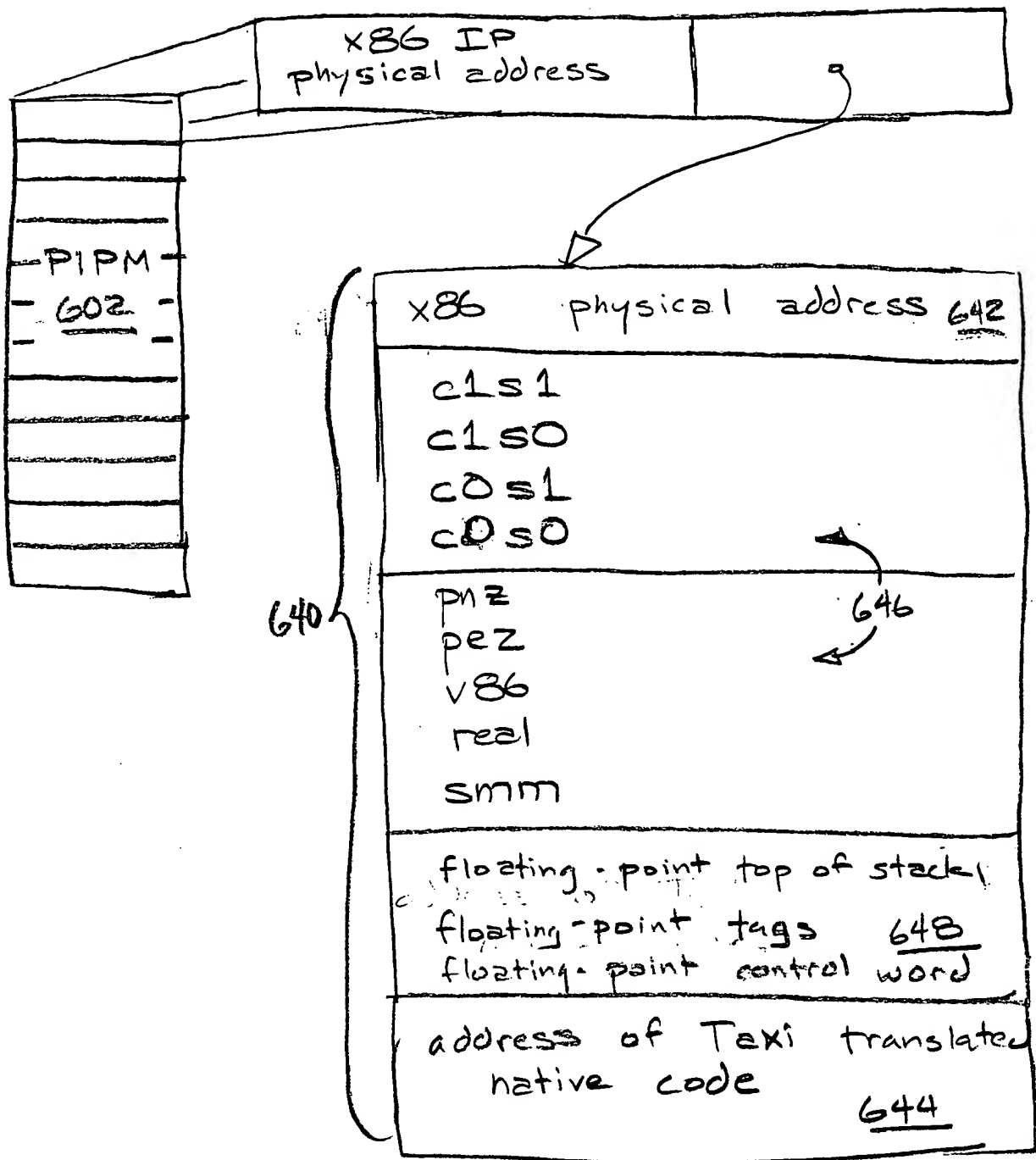


Fig. 6a

Event code from RFE restarting converter
or mapping of converter's x86 opcode

RFE or previous converter cycle

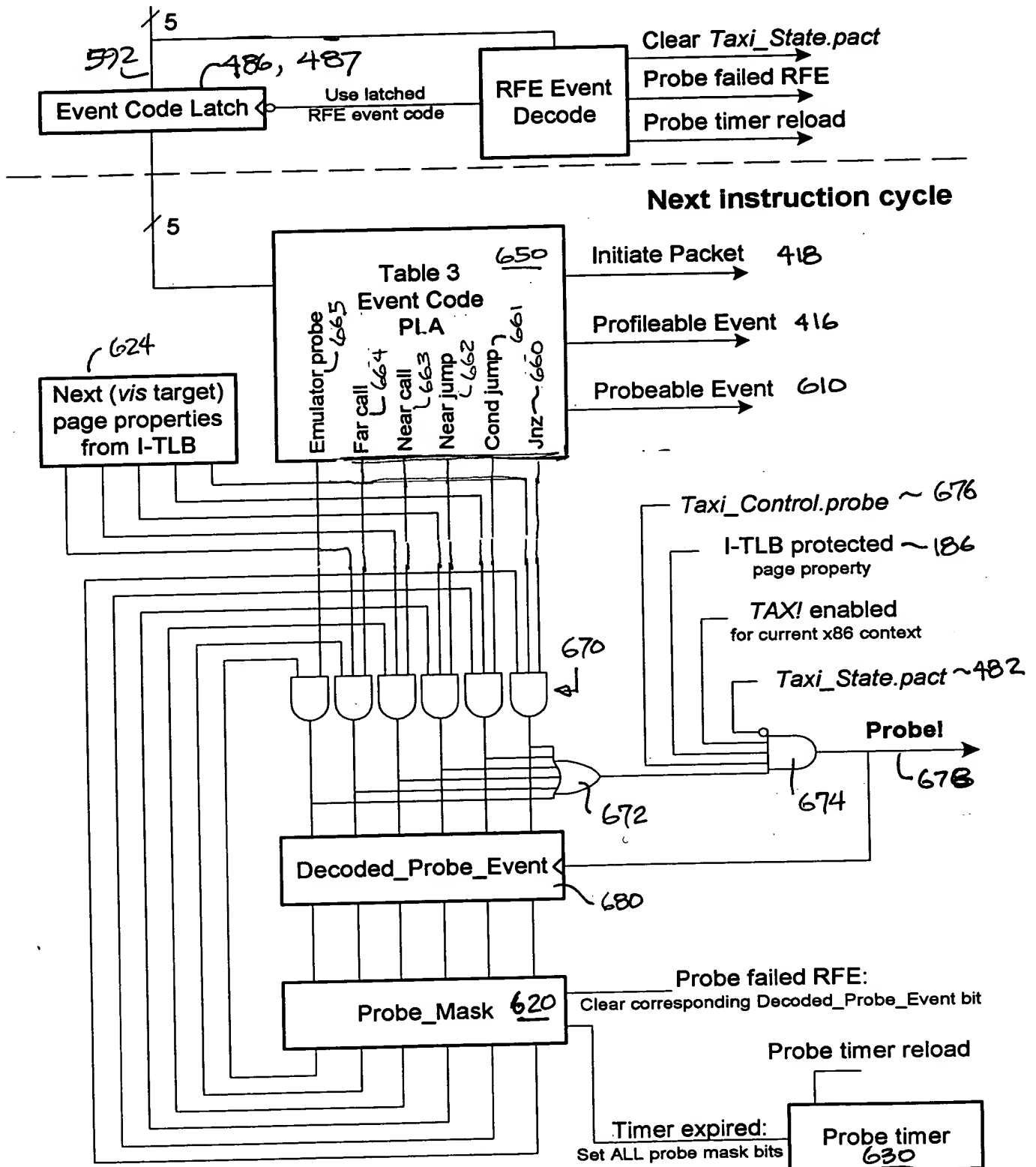


Fig. 6b

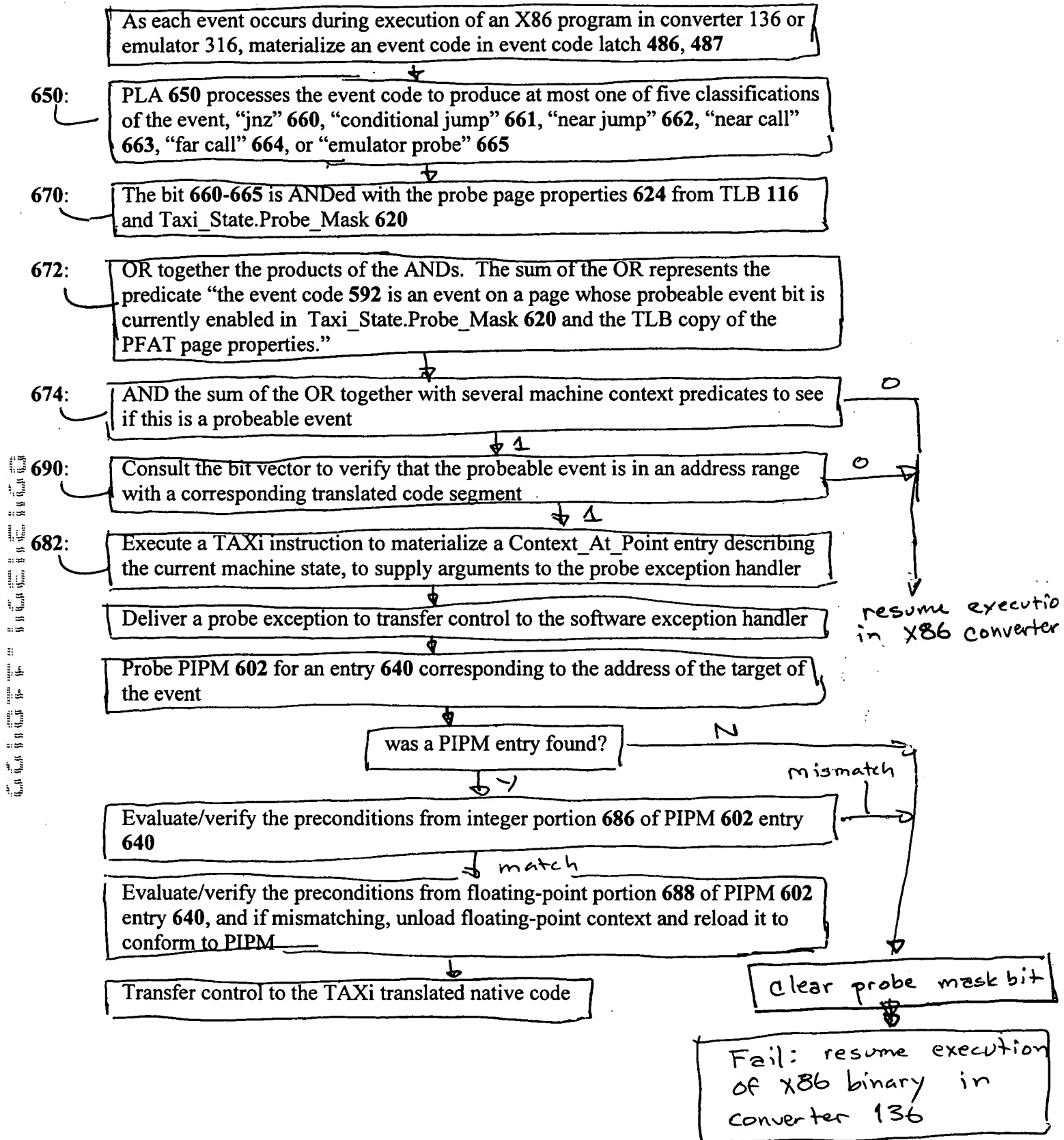
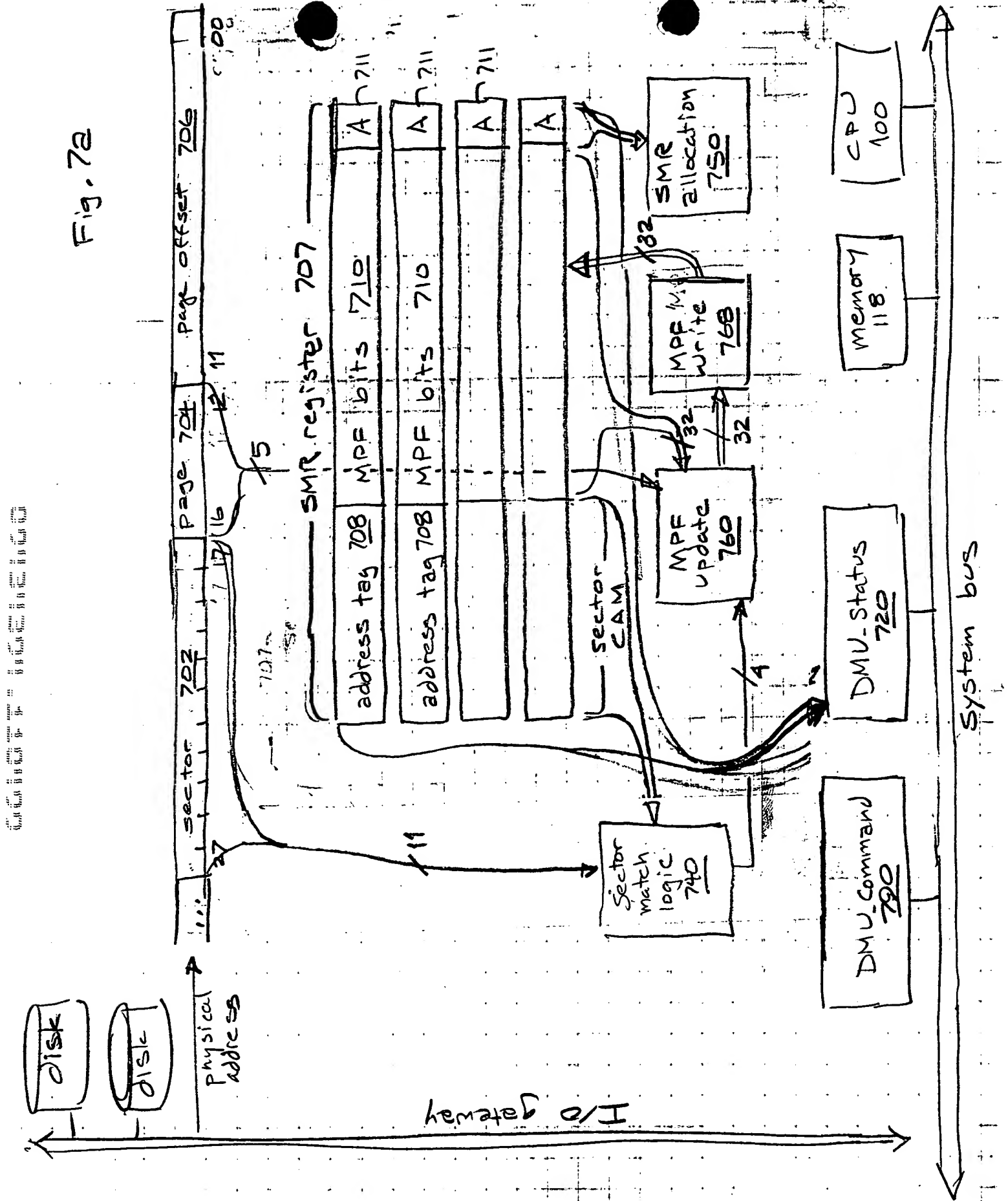


Fig. 6c

Fig. 72



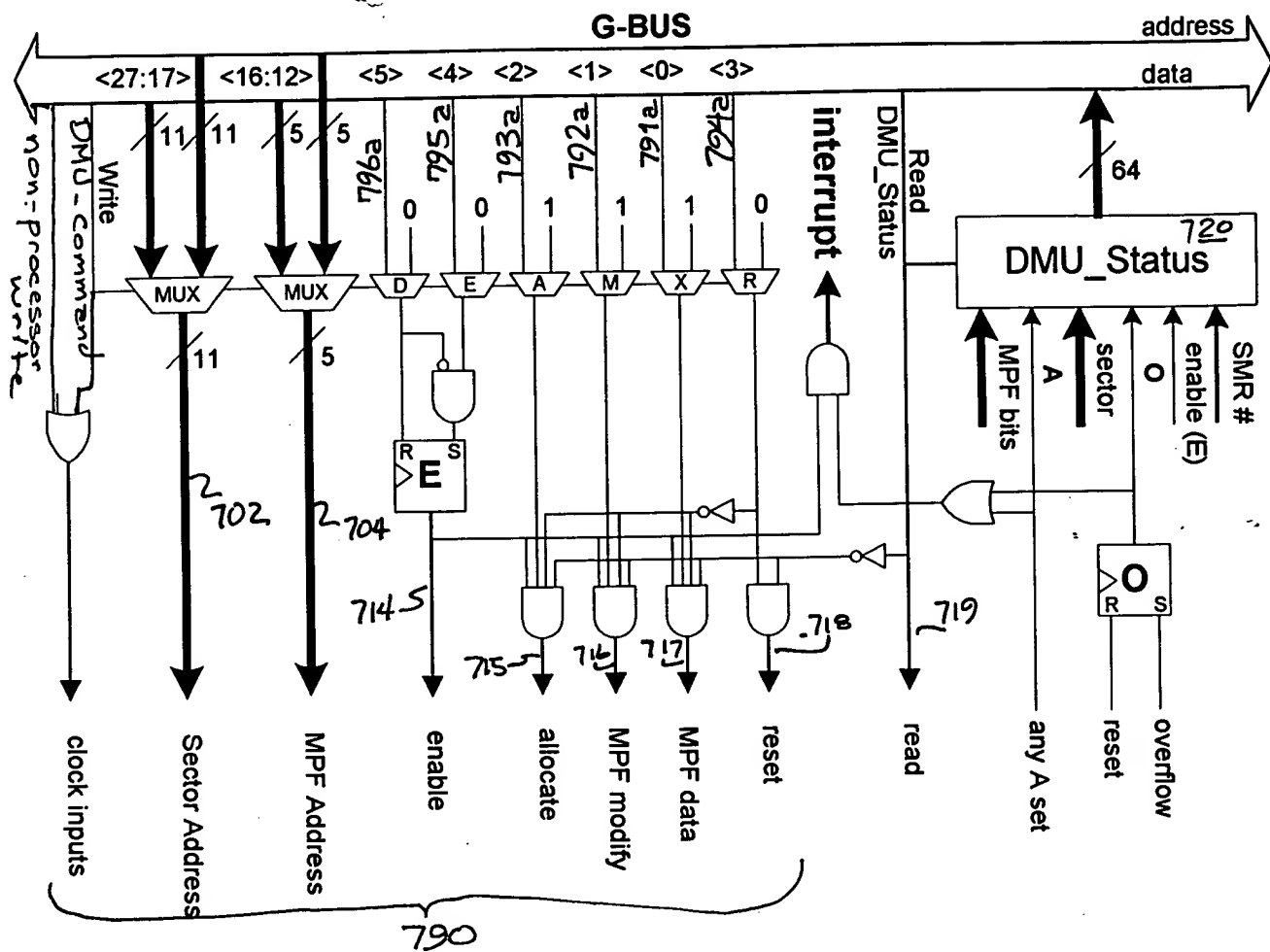


FIGURE 7b DMU interface

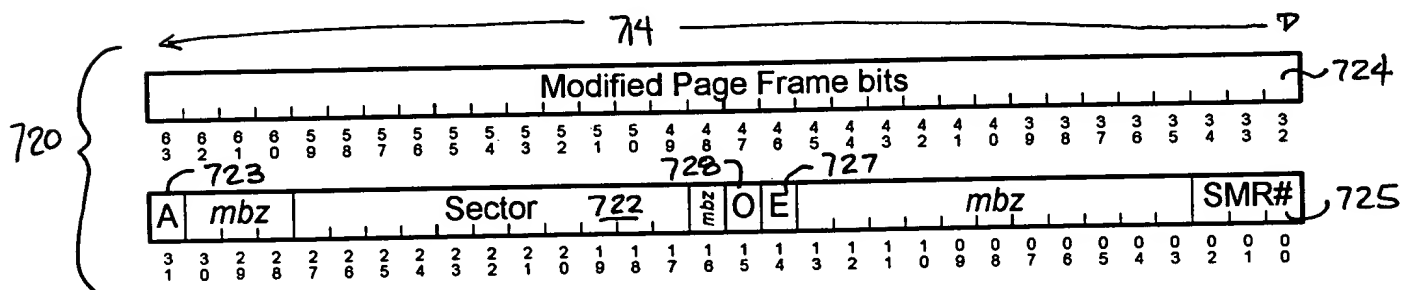


FIGURE 7c The 64-bit DMU Status register

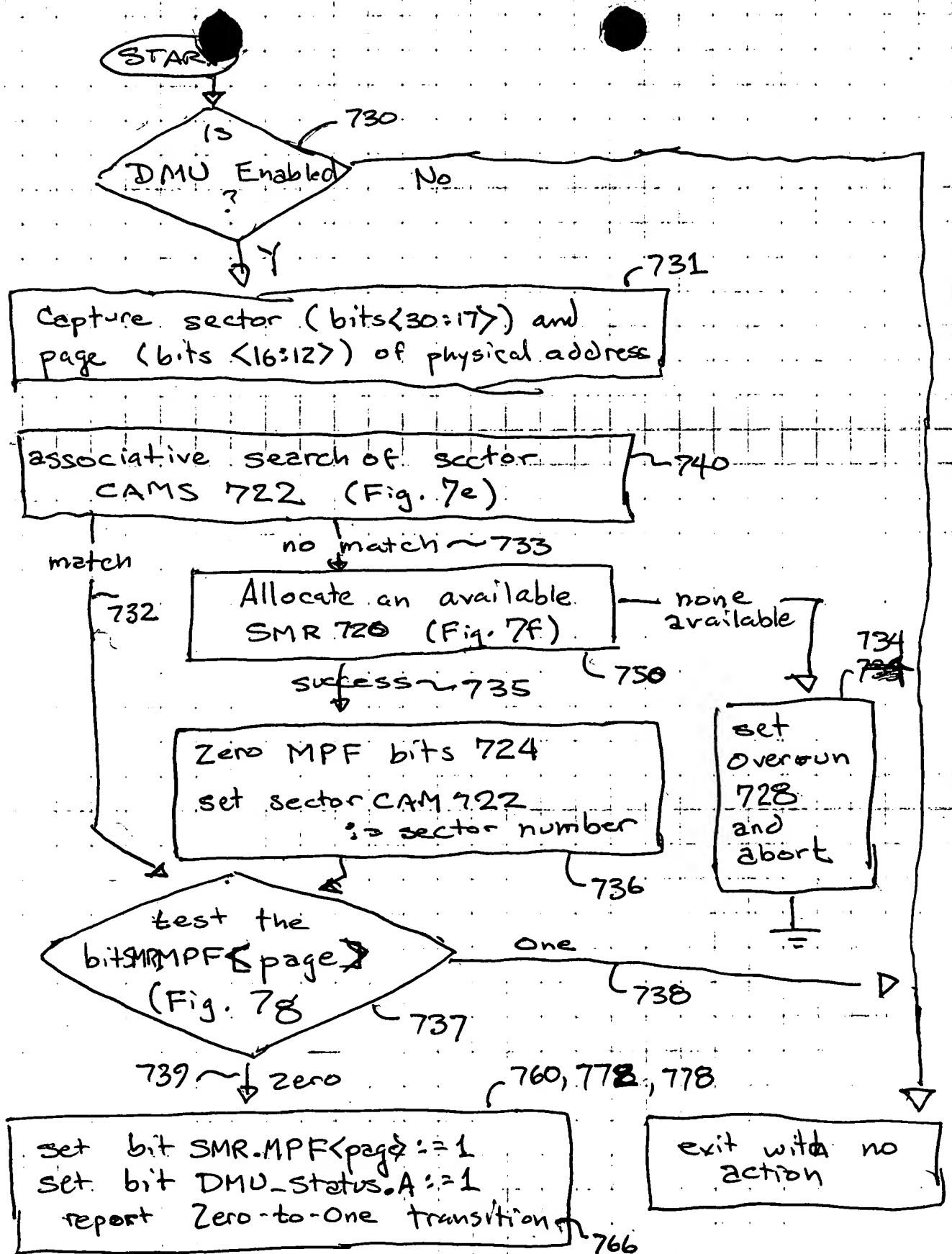


Fig. 7d

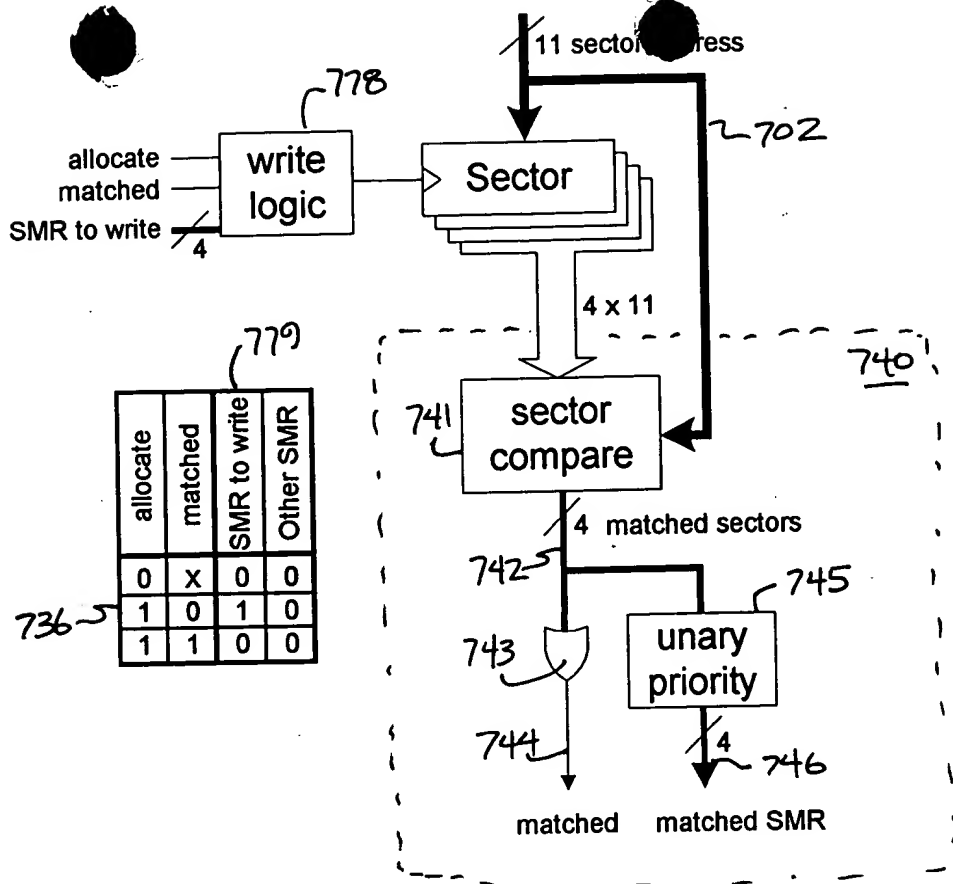


FIGURE 7e Sector match logic

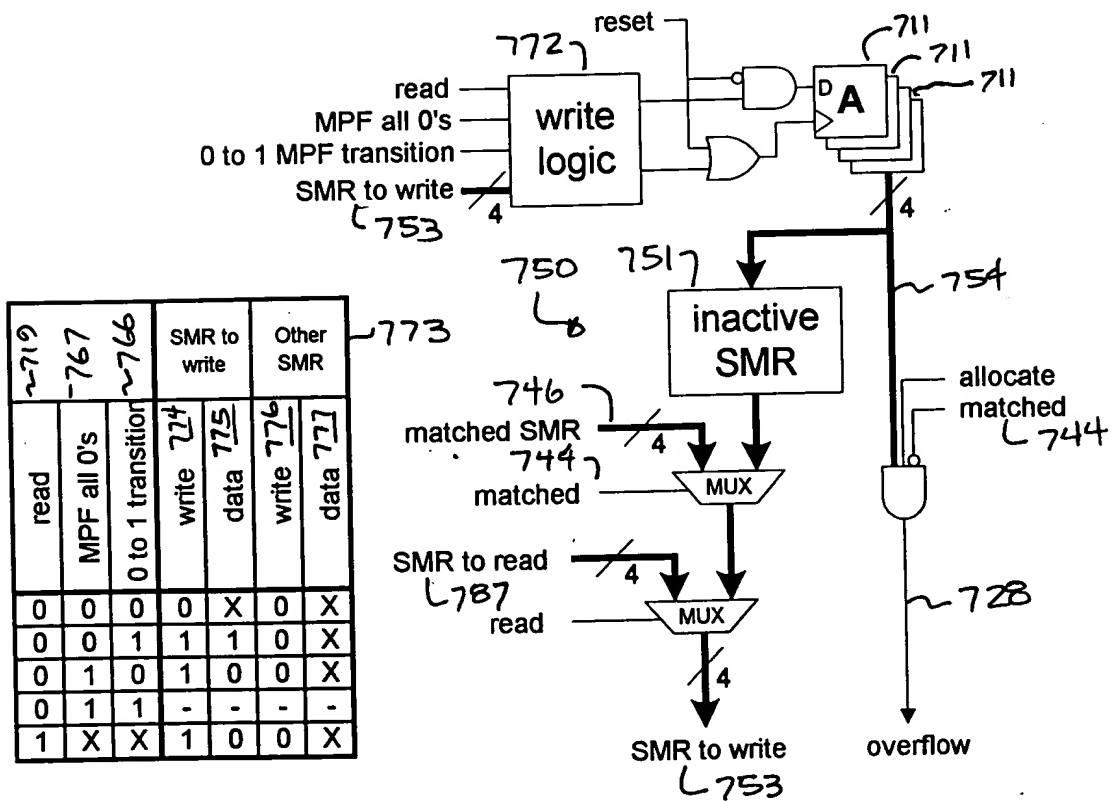


FIGURE 7f SMR allocation

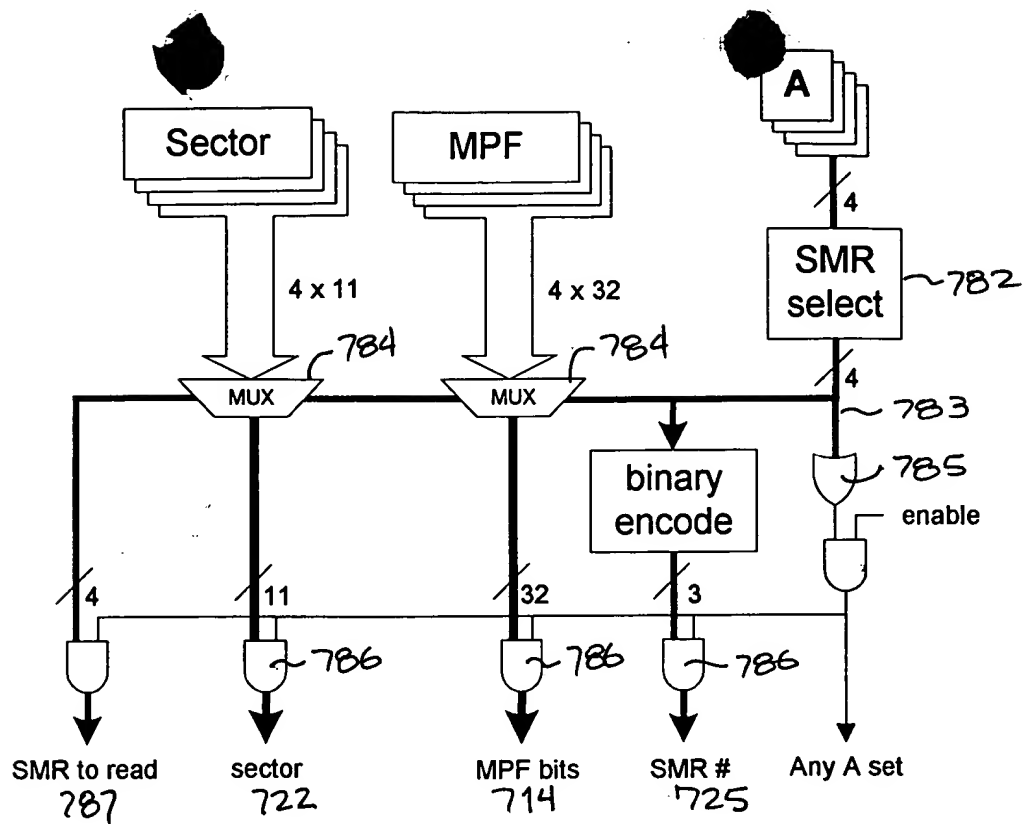


FIGURE 7h DMU_Status read

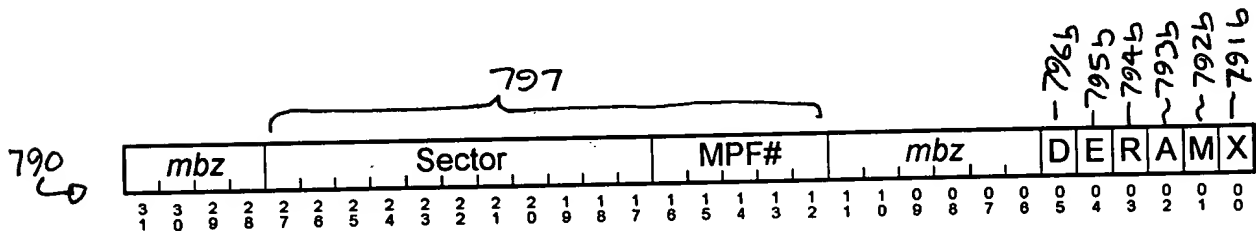
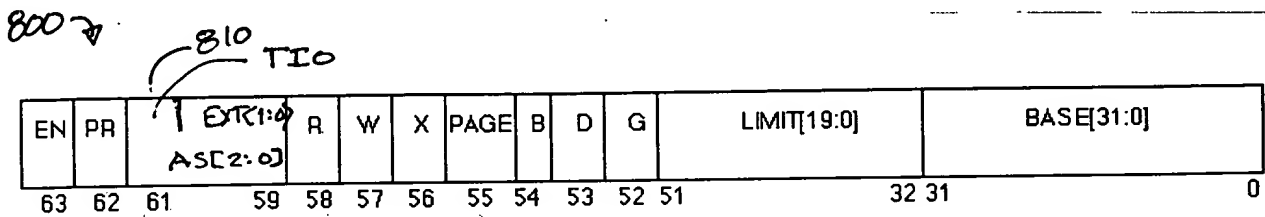


Fig. 7i

Command	Bit	Meaning
Bit	Position	
D	5	Disable monitoring of DMA writes by clearing the DMU Enable flag
E	4	Enable monitoring of DMA writes by setting the DMU Enable flag.
R	3	Reset all SMRs: clear all A and MPF bits and clear the DMU Overrun flag
A	2	Allocate an inactive SMR on a failed search
M	1	Allow MPF modifications
X	0	New MPF bit value to record on successful search (or allocation)

Fig. 7j DMU Commands

M	X	Action
0	-	Inhibit modification of the MPF bit
1	0	Clear the corresponding MPF bit
1	1	Set the corresponding MPF bit



Size	Bit(s)	Name	Function
1	63	SEG.EN	enables segment limit/protection checking
1	62	SEG.PR	chooses which protection bits to use for page table protection - (0 means PSW.UK or 1 means MISC.UK)
3	61:59	SEG.AS	address space (only used when SEG.PAGE is 0)
		SEG.TIO, SEG.EXT	address space extension (only used when SEG.PAGE is 1)
3	58:56	SEG.RWX	read/write/execute '1' means enabled - all 000 means it's an invalid segment
1	55	SEG.PAGE	enables the paging system -- (translation and checking)
1	54	SEG.B	segment size (1 means 32-bit, 0 means 16-bit)
1	53	SEG.D	segment direction (0 means expand up)
1	52	SEG.G	size of limit (1 means it's in 4k pages)
20	51:32	SEG.LIMIT	segment limit
32	31:0	SEG.BASE	segment base

Fig. 8a

At code generation time:

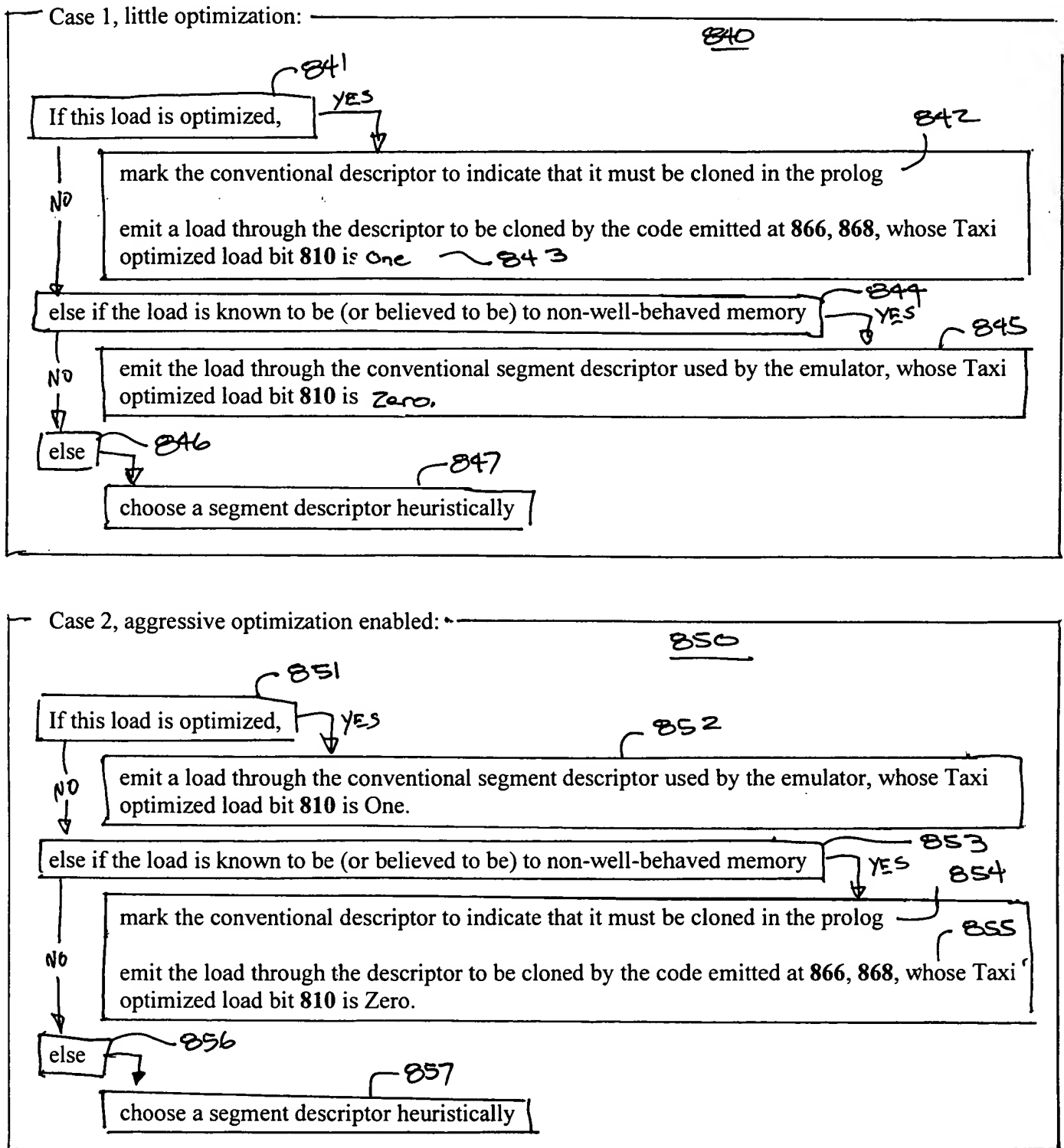


Fig. 8b

TAXi code prolog generation by TAXi translator

860

862 For each native X86 segment descriptor:

done

864 If this descriptor is marked to indicate that a cloned copy is required (reflection both optimized and unoptimized references through this segment descriptor) then

else

866 Emit code to copy one of the X86 segment descriptors to one of the segment descriptor registers reserved for TAXi code. The TAXi optimized load bit 810 of the segment descriptor is guaranteed to match TAXi_Control.tio 820

868 Emit code to explicitly set the value of the cloned descriptor's TAXi optimized load 810 to the opposite value.

Emit code to implement the translated hot spot of the X86 code

Fig. 8c